THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité
Informatique
École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Raluca DIACONU

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Passage à l'échelle pour les mondes virtuels

soutenue le 23 janvier 2015

devant le jury composé de :

M. Pierre Sens	Directeur de thèse
M. Joaquín Keller	Encadrant
M. Pascal Felber	Rapporteur
M. François Taïani	Rapporteur
Mme. Cristina Videira Lopes	Examinateur
Mme. Maha Abdallah	Examinateur
M. Sébastien Tixeuil	Examinateur

Université Pierre et Marie Curie

Doctoral Thesis

Scalability for Virtual Worlds

Author:
Raluca Diaconu

Advisors:
Prof. Pierre Sens
Dr. Joaquín Keller

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

Contents

List of Figures

Chapter 1

Introduction

Contents

Do you have a scalability problem?

Among its topics, the Stack Exchange forum hosts a game developer's question about when to start thinking about scalability [36]. User Rits has a common problem. He is about to launch a multiplayer game for which he is *concerned* that it will be *too popular*. His single server solution is strongly limited in the number of incoming users.

The answers reveal how the scalability problem is usually taken care of in such situations.

"Right now, you have zero users, and scalability is not a problem. Ideally, you want to reach the point where you have millions of users, and scalability becomes a problem. [...] Right now, your main concern is to ship. What happens after that; well, you can worry about that later" user tdammers says, in by far the most popular answer.

Bryan Oakley adds "There's no reason to optimize until you know optimization is needed."

Additionally, user dietbuddha replies advising him to "think about scalability before the first line of code is written" and has been met with criticism. "This is terrible advice. There is enough to think about [...] scalability should be the last item." says user btilly.

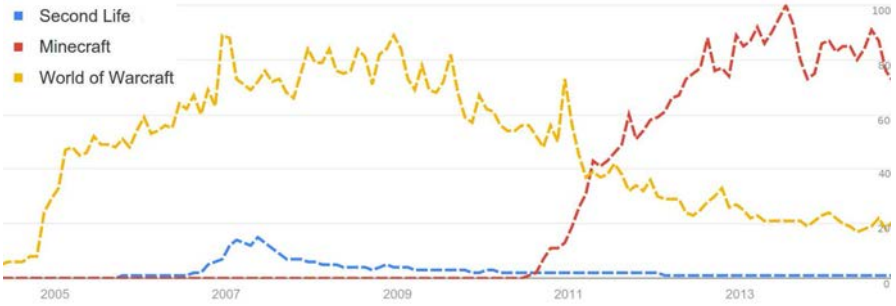


FIGURE 1.1: Interest in major virtual worlds.
Normalized values of the number of Google searches

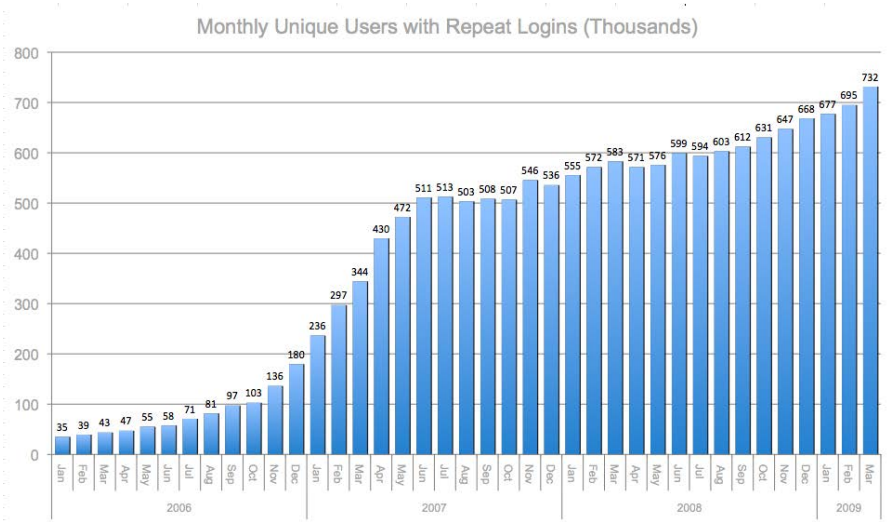


FIGURE 1.2: Second Life unique visitors [32]

A myriad of services dealt with this problem and, just because most of what we see today does not ostensibly show the scalability problem, does not mean this problem does not exist. In fact, these services are popular now just because, one way or another, they figured out how to cope with it *at the right time*. Let's see why this is crucial.

At the moment when social applications become popular, they have an exponential growth. The virality over the internet makes that, in a matter of days, the number of users increases manifold at an exponential rate. Here are some examples.

Figure 1.1 puts together data representing the interest over time in the number of Google searches. World of Warcraft [37] becomes highly popular in 2006 when the interest grows seven fold in just a few days. Similarly, Minecraft [28] went from nothing to huge interest in a very short period of time around 2011.

The initial exponential growth for Second Life [32] in 2006 and early 2007 is clearly illustrated in Figure 1.2. So, when user tdammers says that "You don't go from zero users to ten million overnight" he misses precisely this initial exponential growth. Therefore, coping with scalability on-the-run, when the time comes, is infeasible.

The tricky thing about the scalability problem is that having it usually means your application is popular, appealing to people and –very importantly– with potential revenue. This leads to the general desire to deliver early and get user feedback as soon as possible.

The fallacy arises from the fact that, to attain the revenue and the popularity, the application must be built to last, and therefore, in a way which can handle those millions of users right from the start. Otherwise everybody will leave. Therefore, if you don't get it right at the start, you will never succeed.

In that respect, the scalability problem must be solved independently and ahead of time. Better yet, simple and transparent solutions could be provided once and for all, so that the scalability does not have to be dealt with each time.

This is what we aim for.

1.1 Problem statement

Virtual worlds are computer simulated environments where the users are represented by avatars, in order to see and interact with others. They are also called *networked virtual environments (NVEs)*, because they connect people across the Internet, or *massively multiplayer online worlds*, because gathering many users creates a social setting. They include massively multiplayer online games (MMOGs such as Minecraft [28], World of Warcraft [37], EVE Online), virtual worlds (Second Life [32]), and newly mixed or hybrid reality applications (HybridEarth [18]). Most certainly, Rits' multi-player game discussed on Stack Exchange fits this category.

Virtual worlds attract millions of users and these popular applications –supported by gigantic data centers with myriads of processors– are routinely accessed. However, surprisingly, virtual worlds are still unable to host simultaneously more than a few hundred users in the same contiguous space.

We asked ourselves why. Since the early 1970s, when the first multi-user graphic online game appeared, the per-region algorithmic complexity of running a virtual world is quadratic in the number of connected users to a region: Each event must be sent to everyone.

We have seen that to attain success, it is crucial to scale the world in the number of users, more precisely, avatars. Hence, most solutions rely on some space division. But the key point is to build a system capable to host more users, rather than have more land. After all, it has been shown that terrain is not a problem because only 30% is occupied [109]. However, position updates are the most frequent source of state change.

To relax the bottleneck that impedes avatar scalability, we identified some requirements:

- Separation of virtual world components. To focus precisely on avatar scalability, avatar position updates should be decoupled from the rest of the world. The same holds for objects, or any scarce virtual world aspect. They should be treated separately.
- Geographic independence. Avatars and objects should not be tied to the geographical space they are located in. In other words, avatar mobility and distribution, or object density, should be independent from the terrain size [65].
- Dynamic allocation of resources. As one machine eventually reaches its limits, to be able to scale, we must distribute the load. For a massively distributed system it is necessary to scale up and down dynamically by adding and removing unused resources on-the-fly [76].
- Linear complexity. Informing all avatars about all events occurring in the world is not feasible in a distributed system. The updates received by a user must remain roughly constant.

Finally, to be widely accepted, the solution must be user-friendly and transparent to the developer. For this purpose, one can use the vast and rather underutilized resources offered by cloud facilities. However, so far no current solution addresses these four requirements.

1.2 Contributions

Here we briefly describe our contributions:

- We first performed an evaluation of centralized implementations for two main data structures used for virtual worlds: Delaunay triangulations and R-trees [KDV12]. We focus our scalability analysis on how they handle the trade-off between the number of avatars (and objects) represented and the required frequency of updates.
- To enable scalability in a contiguous virtual space, we propose to separate interest management for the virtual world components, avatars, objects, and terrain. The terrain is static, usually very simple, and does not raise a scalability problem as it can possibly be maintained by every user. Our goal is therefore to provide independent, specific solutions to allow unlimited numbers of avatars to populate the same contiguous virtual world and to handle moving objects, updating their positions with arbitrary high frequencies.

- The main contribution of the thesis is Kiwano [DK13, DK15], a distributed system enabling an unlimited number of avatars to simultaneously evolve and interact in a contiguous virtual space. In Kiwano, we employ the Delaunay triangulation to provide each avatar with a constant number of neighbors independently of their density or distribution. The avatar-to-avatar interactions and related computations are then bounded, allowing the system to scale. The load is constantly balanced among Kiwano's nodes which adapt and take in charge sets of avatars according to their geographic proximity. The optimal number of avatars per CPU and the performances of our system have been evaluated simulating tens of thousands of avatars connecting to a Kiwano instance running across several data centers.

- We also propose Kwery, a distributed spatial index designed to perform efficient reverse geolocation queries on large numbers of moving objects updating their position at arbitrary high frequencies. In Kwery, we use a distributed spatial index on top of a self-adaptive tree structure. Each node of the system hosts and answers queries on a group of objects in a zone, which is the minimal axis-aligned rectangle. They are chosen based on their proximity and the load of the node. Spatial queries are then answered only by the nodes with meaningful zones, that is, where the node's zone intersects the query zone.

- To our knowledge, there is no cloud infrastructure for virtual worlds yet, capable to provide massive, unlimited, scalability. Though, cloud solutions offer vast resources and are progressively being adopted. With Kiwano and Kwery our intention is to provide the first massively distributed and self-adaptive solutions for virtual worlds suitable to run in the cloud.

- For a common use case, when players cannot modify the map, we have designed Manycraft [DKV13], a distributed architecture to scale the number of users together in the same Minecraft map. Minecraft protocol messages are of three kinds: control, entity and map. In this approach, messages are processed by a Minecraft server assigned to the player. Kiwano takes care of the entity related messages and provides to each player the neighbors according to their position.

- By transferring the load generated by the moving avatars to Kiwano, we design OneSim [DK14], a distributed system to allow an unlimited number of users to be together in one contiguous Second Life region. Each user runs an OpenSim instance of the same region, which is populated with their respective neighboring avatars provided by Kiwano.

- HybridEarth [dCDKT14] is a social mixed reality world connecting avatars and people in the same contiguous space: the Earth. The two applications, web and

mobile, have been developed independently in our team and interoperability is ensured through Kiwano. With Manycraft and HybridEarth we show general interoperability for virtual worlds by taking care of avatar positions separately.

- In Kiwano, we define a new neighborhood relationship for avatars based on Delaunay triangulation graphs. The number of neighbors is bounded by the degree of the chosen graph, thus providing linear complexity.
- Kwery comes with a model for reverse geocoding that can be applied for any range-query, not only geographical.

To support these solutions, we have developed within our team the Manycraft implementation, two HybridEarth applications (web and mobile), and a test Kwery version with performance evaluation. These implementations are mainly the work of my colleagues. My contribution to these actual implementations is limited. For the sake of clarity, I will remind the contribution along the thesis, when necessary.

1.3 Outline of the thesis

This thesis report comprises three parts. In the first one (Chapters 2 and 3) we present the background notions and an analysis of state-of-the-art solutions for virtual world scalability. In the second part (Chapters 4 and 5) we propose our twofold contribution, Kiwano and Kwery, respectively. In the last part (Chapters 6, 7 and 8) we detail how we employ our Kiwano architecture to design three different types of virtual worlds: Manycraft, an MMOG, Second Life, a virtual world, and HybridEarth, our mixed reality world. We show, for each of them, how Kiwano addresses the scalability problem and other particular requirements.

Altogether, the manuscript is organized as follows:

- Chapter 2: Background. In the first chapter we present the concept of virtual worlds and explain that avatar movement is the main cause of the scalability problem. The goal is to provide the reader with an adequate background to fully understand the context and the contribution of the thesis.
- Chapter 3: State of the Art. We begin by analyzing the limits we attained by measuring centralized implementations for the two main data structures to scale virtual worlds, Delaunay triangulations and R-trees. Then, we investigate some state-of-the-art solutions for virtual world scalability.

- Chapter 4: Kiwano: Avatar Scalability and Neighborhood Updates. Our first and most important contribution is Kiwano. In this chapter we introduce the notion of neighborhood based on the Delaunay graph. We show how to distribute the set of avatars dynamically, in a peer-to-peer fashion among the nodes of the system. We evaluate the current implementation with tens of thousands of avatars and propose a transparent API to the developer.

- Chapter 5: Kwery: Spatial Containment Queries for Moving Objects. In this chapter we present Kwery, a distributed system designed to efficiently perform axis-aligned range queries on large numbers of highly dynamic objects updating their positions up to several times per second. We describe our preliminary results.

- Chapter 6: MMOG Case Study: Manycraft. We present the scalability limits of a Minecraft world and analyze their source. We propose Manycraft, a scalable distributed architecture on top of Kiwano, and describe the implementation.

- Chapter 7: Virtual World Case Study: OneSim. Using a similar approach, we discuss in this chapter how to build OneSim, a distributed architecture to allow an unlimited number of avatars inside the same Second Life region.

- Chapter 8: Interoperability for a Shared Hybrid Reality: HybridEarth. We present HybridEarth, a mixed reality application, and how the two worlds, the mirror world and the augmented reality, are interconnected with Kiwano.

- Chapter 9: A 3 Point Perspective. We conclude the present thesis by looking at our work from three points of view: what we accomplished, what future perspectives it opens, and what remains to be done.

1.4 List of international publications

[dCDKT14] Jean de Campredon, Raluca Diaconu, Joaquín Keller, and Elodie Triponez. Hybridearth: Social mixed reality at planet scale. In *CCNC'2014 - Demos*, Las Vegas, USA, 2014.

[DD12] Raluca Diaconu and Catalin Dima. Model-checking alternating-time temporal logic with strategies based on common knowledge is undecidable. *Applied Artificial Intelligence*, pages 331–348, 2012.

[DK13] Raluca Diaconu and Joaquín Keller. Kiwano: A scalable distributed infrastructure for virtual worlds. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 664–667, 2013.

[DK14] Raluca Diaconu and Joaquín Keller. Onesim: Scaling second life with kiwano. In *Proceedings of International Workshop on Massively Multiuser Virtual Environments*, Singapore, 2014.

[DK15] Raluca Diaconu and Joaquín Keller. Kiwano: Scaling virtual worlds. In *Submitted to International Conference on Distributed Computing Systems*, 2015.

[DKV13] Raluca Diaconu, Joaquín Keller, and Mathieu Valero. Manycraft: Scaling minecraft to millions. In *NetGames 2013*, Denver, USA, 2013.

[KDV12] Joaquín Keller, Raluca Diaconu, and Mathieu Valero. Towards a scalable dynamic spatial database system. *CoRR*, abs/1211.4414, 2012.

[TD14] Ferucio Laurentiu Tiplea and Raluca Diaconu. Petri net computers and workflow nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2014.

[VDK13] Mathieu Valero, Raluca Diaconu, and Joaquín Keller. Manycraft: Massively distributed minecraft. In *NetGames 2013 (Demo)*, Denver, USA, 2013.

Part I

State of the Art

Chapter 2

Background

Contents

Mixed reality, massively multi-user games, virtual worlds, cyberspace are among the most alluring concepts in computer science. Less alluring is however the notion of *scalability*. It is hard, it is also often avoided because it is seen as premature optimization[1]. In this first chapter we try to immerse the reader into the world of virtual worlds but with the clear intention to understand the scalability problem. The goal is to provide the reader with an adequate background to fully understand the context and the contribution of the thesis.

We begin the thesis with a taxonomy of worlds, from virtual to real, the alluring side of computer science. We discuss what distinguishes them and what brings them to our focus in Section 2.1. In Section 2.2 we identify the scalability problem to be a major obstacle

[1] "Premature optimization is the root of all evil", Donald Knuth, 1974

in the development of virtual worlds. We identify avatar mobility and distribution to be an important factor and analyze the challenges. The rest of this chapter, Section 2.3, lists the general notions and conventions used along this thesis.

2.1 A taxonomy of worlds

Immersion in a virtual reality was initially understood as completely decoupled from the real world. We had games that gathered together people in a unique virtual space. Then we had customizable virtual environments, where interaction had no more a predefined purpose encoded in a gameplay, and allowed users to behave freely and build their own world.

With the technological advances we became more and more aware of the reality, we were able to reproduce it into mirror worlds. This makes it possible to have a hybrid world, half real, half virtual, with avatars and people side by side in the same shared space.

2.1.1 Massively multiplayer online games

Maze War [60] in 1974 connected for the first time –through ARPANET– several players across the US in a unique virtual space. It is the game that introduced several concepts at the base of today's virtual worlds, *avatar*, first-person 3D perspective, position on a map, multi-player network game, various interactions between players (chat, shooting).

Nowadays such games are called Massively Multiplayer Online First Person Shooters (MMOFPS) and are large-scale virtual battlegrounds. Avatars have fast and precise movements and interact often. There aren't many objects, most of them are weapons and items owned by avatars. Initially, designed to support only up to 64 players together, games as Quake [31] and Counter Strike [13] were not classified as massively multiplayer. Recently, they have been used as research tools for designing scalability [46, 88]. Tanks and robots uses Pikko Server to attain the world record of 1,000 players together [38].

Massively Multiplayer Online Role Playing Games, or MMORPGs, such as World of Warcraft [37], offer a rich environment, with detailed avatar customization, complex modifiable objects and non-player characters (NPCs). Interactions involve avatars, NPCs, and objects in the environment, implying high requirements to simulate the world.

Ultima Online [34] was the first MMORPG to offer a persistent world and to reach the 100,000 subscriptions, exceeding by far any game that went before it [34]. However, the number of players together in the same space has the same limitations.

2.1.2 Metaverses

Taking inspiration in the '80s science fiction literature, multi-user virtual worlds were designed to be fully customizable, highly interactive environments. Second Life [32] is the most successful virtual world with a million active users logging in and inhabiting the world every month [104]. It is mostly used for social activities, thus avatar-to-avatar interactions are prevalent and the scalability is of utmost importance.

In this area scalability is a well identified problem [77, 110]. For Second Life, research has been conducted to understand avatar behavior [88, 110], and several solutions to scale have been proposed [90, 91, 107].

OpenSimulator [29] is a free and open-source software to develop virtual worlds compatible with Second Life clients, which attracts scalability research [87, 90, 91]. By distributing the world state among the users in a peer-to-peer architecture, Solipsis [83] is the first virtual world that can be theoretically inhabited by an unlimited number of participants.

2.1.3 Mirror and hybrid worlds

Back in 1993, before the commoditization of spherical cameras, rotating lasers for 3D scanning, antennas to map the Wifi hotspots, and other sensors to map the physical world, the term *mirror world* was coined [74]. It refers to an accurate representation of the real world in digital form. Google Street View [35], Google Earth [17], and Bing Maps [10] are among the generally known examples.

So, as these copies of the real word resemble it more and more, they are still empty, we do not see people roaming in the streets, or other users looking at the same places we are. Prototypes [61, dCDKT14] show the feasibility of a hybrid world using augmented reality and virtual spaces. Indeed, 7 billion potential users in the same virtual world exceeds by some orders of magnitude the state-of-the-art.

2.2 The scalability problem

To refer to all these systems we will use the general term *virtual world*, for simplicity. In the literature they are also called *networked virtual environments (NVE)* [57, 81, 89]. Let's break them down into components and isolate causes for the scalability problem. This separation of concerns will help us provide specific solutions for each aspect that comes into play.

2.2.1 Context

All the aforementioned applications involve a virtual space, being a battlefield, an always changing place entirely made by its users, or even a virtual world, copy of the real world. Avatars, alter-egos of their users, inhabit these spaces and interact with the environment and with one another. So, virtual worlds are usually composed of [85]:

Avatars are characters controlled by players. Their representation can vary from just a point with position coordinates to fully customizable and detailed with visual appearance, health and objects.

Non-player characters (NPC) look and behave as avatars but are controlled by an algorithm.

Mutable objects are objects that can be moved and modified by avatars or NPCs. They must support frequent and sometimes concurrent reads and writes.

Immutable objects usually describe the terrain. They form the read-only landscape information.

To be present in a virtual world, the user is represented by an avatar. With it, the user interacts, changes the world state, and receives information relevant to his avatar's view of the world. There are several things an avatar can do:

Avatar updates such as position updates, animations, skin modifications, but also interactions with other avatars like chat.

Objects updates occur when the avatar manipulates, creates, or destroys some object.

Data streams are directly sent to one or a group of interested avatars, like real-time audio and video.

The server hosting the world, *i.e.*, the simulator, receives updates from the users, then decides which actions are allowed and computes their effect. Finally, it informs each user about the changes that are of interest. This computation is costly, making the maintenance of the world quadratic in complexity.

These actions performed by the users in a virtual world may be interchangeable or not.

Commutative when the order does not matter;

Non-commutative when performing two operations in a different order will produce a different world state.

For instance, if Alice and Bob kiss each other in a short time frame, it is of little importance who kissed first. After a short delay the two possible action orders will lead to the same world state. However, if Alice and Bob shoot each other, the order will dictate who is alive at the end. The resulting world states will be inconsistent with each other.

To have everyone in the same shared virtual space, users connect to the same server that simulates the world. At some point, when the number of connected users increases, the server runs out of resources. So, we need to add more servers, we need a distributed system.

Since they are read-only, scalability for immutable objects can be ensured by simple replication on multiple servers or on the clients. Scalability becomes an issue when data is frequently changed and intensively accessed. Scalability must allow more objects, more avatars and seamless interactivity.

The current state-of-the-art solutions attempt to provide scalability for the entire virtual world. Approaches to scalability treat all these components at the same time. This monolithic view of virtual worlds fails to take into account that various components have different requirements. Object management needs to provide persistence while avatars need high responsiveness. Also, a complex décor must not impede avatar interaction, as it happens most of the time in Second Life [109].

Hence, this defines the scalability problem in virtual worlds.

We are interested in providing different solutions for avatars and dynamic objects. We discuss the different requirements in detail in Section 3.2 and mostly when introducing our contributions, see Chapters 4 and 5.

The most prevalent actions in a virtual world are position updates, known to represent about 70% of the traffic [57]. The good news is that they are commutative and thus, facilitate the distribution of the world simulator in an asynchronous system.

2.2.2 Avatar mobility

The bad news is that these position updates trace unpredictable avatar distribution and mobility patterns.

A number of works [88, 89, 109] inferred from Second Life traces that avatar distribution forms a few high density hotspots leaving most of the space nearly desert. The blue banana model [88] states that avatars spend most of their time moving slowly and

erratically around hotspots. From time to time, avatars travel long distances until they reach another hotspot.

These observations match the power law distribution of human populations [75, 112], with people concentrated in urban areas. At a smaller scale, in cities, points of interest have higher densities. Furthermore, these observed points of interest change during the day. Moreover, it has also been observed [112] that human mobility occurs in bursts. Most of the time people make small movements while only occasionally they move straight, for a long distance.

All in all, the density follows a power law distribution for both, humans and avatars, while mobility occurs in bursts.

2.2.3 Chalenges

The frequent avatar position updates stress the world simulator as they have to be timely sent to all interested users. More than this, the unpredictable avatar movement makes distribution the most challenging task to attain scalability.

This problem has been addressed in two ways:

- To reduce the load for the client, events are assumed to have only a local effect. As users are concerned with what is happening nearby, they need to be notified of all the events produced by those located in their neighborhood, that is, a zone around their position that will provide the complete scene, the visible décor and all the events. But this does not take into consideration variations in density. An area of interest with a fixed size often can be too large, when there are many people around, or too small, where the density is lower. Also, if avatars have areas of interest of different sizes, this can produce asymmetry: you see your neighbor, but she does not see you. Therefore, one challenge is to find a good way to connect nearby avatars.
- One machine eventually reaches its limit, so the simulator needs to be distributed. But to compute for each update who is concerned is costly, quadratic in complexity. Most solutions rely on fragmenting the world and distributing contiguous parts of the virtual space. The load is handled by more machines, but it is not distributed evenly because with avatar mobility the distribution changes over time. More, a challenge is how to efficiently distribute the load in a self-adaptive way.

The envisaged applications have high requirements in terms of responsiveness. The temporal resolution can be as high as 10 or 30 frames per second. The CAP theorem [51] states that an application must chose two out of three: consistency, availability, partitioning. This principle has been applied to virtual worlds as well [64, 65] so, a distributed world cannot be both highly available, namely, responsive, and strongly consistent. A scalable system must therefore address the trade-off between consistency and responsiveness.

2.2.4 Scalability beyond virtual worlds

Our contribution presented in this thesis is targeted at scaling virtual worlds, and we present applications for games (Chapter 6), metaverses (Chapter 7) and hybrid worlds (Chapter 8). But the applicative scenarios that can benefit from this research is broader than this.

Location information management is a widely encountered problem in computing systems. Retrieving which objects are at a specific location or who is nearby is still a challenging issue, especially when dealing with huge amounts of entities and frequent updates. Applications include:

- *Geographic Information System (GIS)* [30] are used to find who or what is at a specific location. Also, they need identifying the nearest points of interest or monitoring various geographically identified targets. For instance, a tourist information system may allow users to search for attractions within their vicinity.
- *Car fleet management*, for companies to know and optimize at any moment how their resources are spatially allocated. In search for a taxi for instance, the system must provide preferably the nearest one.
- *Local advertising* and *georecommendation* [55]: A typical scenario is sending ads to potential customers in the vicinity of a local commerce. For instance, a restaurant with 20 meals left at 1PM decides to send 50%-off coupons by text message to phone users nearby.
- *Location (geo) based social gaming* such as Google's Ingress [21]. Other online and social activities are outdoor geocaching [15] or proximity dating, see Tinder [33].
- *Social mixed reality* [18, dCDKT14]: In virtual worlds avatars are mobile objects and spatial queries are issued to determine which avatars and objects are relevant for a given user. In the case of mixed –a.k.a. hybrid– reality, the system must represent uniformly, virtual and real objects.

All these applications need to deal with large numbers of moving objects in a timely manner. In particular, for some applications, meaningful time intervals are minutes –*i.e.*, advertising– while for others, events might occur every few milliseconds –see virtual worlds and social mixed reality. Also, in some scenarios the number of moving objects does not exceed a few hundreds while in others millions are expected.

A scalable application needs to take into consideration the trade-off between the number of moving objects and the frequency of their update in order to provide an adaptive solution when needed.

To summarize, solutions for the scalability of virtual worlds have applications in many areas dealing with numerous mobile objects with frequent position updates. They all have to provide nearby information or to query data spatially located.

2.3 Notions

Graphs make the foundational abstraction for modelling such complex systems. Indeed, they constitute the model for system architecture, for world state distribution, and, in this thesis, we will propose the avatar neighborhood relation based on Delaunay graphs. Trees –balanced trees especially– have intensively been used in the literature to reduce data accessing from linear to logarithmic.

2.3.1 Delaunay triangulations / graphs

A *triangulation* of a set of points P is defined as a maximum planar graph whose vertex set is P, *i.e.*, no edge connecting two vertices can be added without destroying the planarity. The formal definition of a *Delaunay triangulation (Delaunay graph)* for P, noted $D(P)$, states that no *vertex* in P is inside the circumcircle of any triangle in $D(P)$. Moreover, two vertices form an edge if and only if there is a closed disc that contains the two vertices on its boundary but does not contain any other point of P. Informally, Delaunay triangulations maximize the minimum angle of all the angles, by connecting vertices that are close.

The *Voronoi diagram* is the dual of a Delaunay graph and represents proximity information about the set of objects, see Figure 2.1. The two-dimensional space is partitioned by assigning to each point its nearest object called *generator*, *site* or simply *vertex*. The points whose nearest vertices are not unique will lie on the diagram's edges delimiting the zones of the corresponding object sites [67].

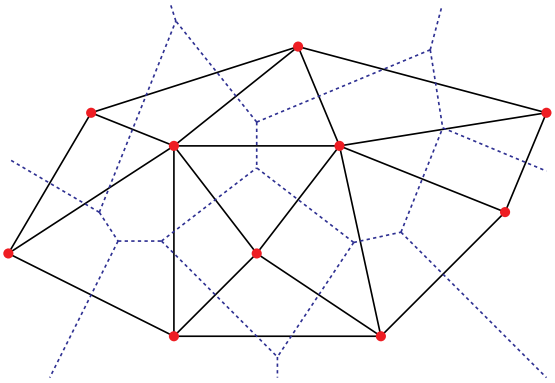


FIGURE 2.1: Delaunay triangulation (solid) and Voronoi diagram (dashed)

Point location consists in line walking inside the triangulation from an arbitrary object to the nearest vertex via the edges. The complexity is $\mathcal{O}(N)$ in the worst case and $\mathcal{O}(\sqrt{N})$ on average when the objects are distributed uniformly at random, where N is the size of P. Any operation on a certain vertex (*e.g.*, insertion, deletion, position update) will first locate it inside the triangulation. Apart from point location procedure, insertion and deletion of one vertex have linear time complexity with respect to N in the worst case scenario, such as when points are circularly aligned. In practice this case has null probability, and a constant complexity is expected when the vertices are distributed uniformly at random, see [67]. The nearest neighbor query is similar to a point location, by performing a line walk to the nearest vertex inside the triangulation. Hence the complexity is $\mathcal{O}(N)$ in the worst case and $\mathcal{O}(\sqrt{N})$ on average. Retrieving the neighbors has a constant cost since each node maintains this information.

Delaunay triangulations in two dimensions are unique. In three dimensions they are not, also the computation is too costly. However, two dimensions are enough to reason about geolocated points. A Delaunay triangulation provides a connected graph of a set of vertices based on their proximity. Furthermore, the average number of edges per vertex is independent of the size of P, or the density, and it is six on average. This property made them a good candidate to model per-to-peer systems [41, 52, 80, 83, 84].

Lemma 1 (Local Delaunay Lemma [58])**.** *Let P be a set of vertices in the plane. If the ordered subset $P' = \{v_1, v_2, ..., v_N\} \subseteq P$ forms a simple polygon containing $v \in P$, then the Delaunay neighbors of v are contained in the union of the circumcircles of the N triangles formed by v and every two consecutive points of P' (irrespective of the triangle orientation).*

In other words, the lemma allows us to distribute the vertices onto many nodes because the interconnected links between nodes remain constant on average. Therefore, if each link comes with costs, this approach states that, if changes are local, the system can, theoretically, scale indefinitely at no additional cost.

For our purposes, every avatar is represented by a vertex in the Delaunay graph with its coordinates in the virtual world. Two avatars are neighbors –or one-hop neighbors– if their corresponding vertices are connected in the Delaunay graph.

2.3.2 Trees and hierarchies

Trees 'sprout' about everywhere in computer science[2] and they have been designed to respond to various querying requirements.

B-trees [40] have been introduced to index data hierarchically to allow search, access, insertion, and deletion in logarithmic time. Originally, the key space was designed with one dimension to be used mainly in relational databases and file systems. *R-trees* [78] are a variant for multi-dimensional key spaces using bounding axis-aligned rectangles.

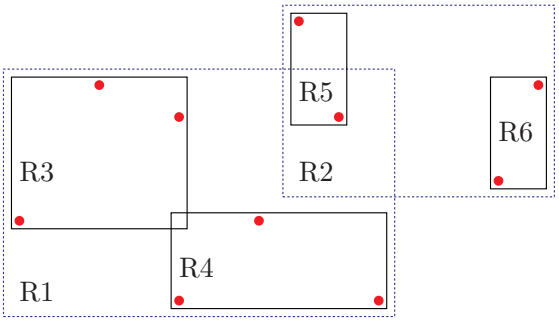


FIGURE 2.2: R-tree example

The nodes of the tree hierarchy store *minimal bounding rectangle* $r = (x_r, y_r, X_r, Y_r)$ with the coordinates of its bottom-left corner x_r, y_r, and top-right corner X_r, Y_r. The nodes form a balanced tree structure where the objects' representations are stored in the leaves. The space required to store the full data structure is linear.

The depth of an R-tree storing N objects is $\mathcal{O}(\log N)$. The location procedure requires $\mathcal{O}(\log N)$ time. The insertion and deletion of a node, apart from the location procedure, must also re-balance the resulting tree. This uses a fixed number of operations so the overall complexity remains logarithmic.

To move an object, we remove it and then insert it at the new coordinates. Finally, the algorithm for range query will use the bounding boxes to decide whether or not to search in a subtree. Therefore, most of the nodes will never be visited resulting in a logarithmic complexity.

In Section 3.1 of the next chapter we provide a scalability study using state-of-the-art implementations for both Delaunay triangulations and R-trees. We focus our analysis on

[2] "Trees sprout up just about everywhere in computer science", Donald Knuth, 2011

how they handle the trade-off between the size of the data structure and the frequency of updates supported for the aforementioned virtual worlds.

Chapter 3

State of the Art

Contents

When working on virtual worlds, as described in the previous chapter, we need to face a huge problem, maybe the most important to date. Today's virtual worlds barely

host hundreds, at best thousands, of simultaneous avatars evolving together in the same contiguous space.

In this chapter we investigate the current state of the art for virtual world scalability. We begin by measuring the limits of current implementations for the two main approaches presented earlier, Delaunay triangulations and R-trees, for a unique machine. Our measurements coincide with the limits of current virtual worlds and other use cases but, more than this, they assess how much better a distributed solution can perform.

Scalability is a major concern in computer science, which has been addressed in many areas. The state of the art in virtual worlds scalability spreads across other related areas, indexing techniques, geo-spatial databases, GIS, to name just a few. In this section we recall and relate to various approaches we deem of interest for the scalability of current and emerging virtual worlds.

As one machine eventually reaches its limits, we need a distributed system to maintain the world state. But since users need to be timely informed about the changes, the distribution is not an easy task.

- The first step for scalability is to reduce the complexity of maintaining the world state to sub quadratic. This has been addressed with interest management methods, by limiting the visibility of events and users' view capacities.
- As one machine eventually reaches its limit, we need to have distributable data structure in which the access (latency) does not increase (significantly) with the size of the world. This is the second step to scalability.
- The third step is to ensure efficient algorithms to maintain the data structure practically feasible. For scalability, the system architecture needs to make sure the needed resources are easily accessible when needed.

This chapter is structured accordingly. We first performed an evaluation of centralized data structures suitable for virtual worlds, and our results are presented in Section 3.1. In the following section we review the interest management techniques, see Section 3.2. Then we list the most prevalent structures for data distribution, see Section 3.3. We emphasize their distributable nature and how much they are able to answer the interest management requirements. Next, in Section 3.4 we present the architectural design and the practical scalability of actual systems. We end this chapter with our remarks on what the current state of the art lacks in order to provide further scalability and how we plan to bring our contribution.

In addition, Kiwano and Kwery, the scalable solutions we propose for virtual worlds, are introduced with the same presentation structure in Part II.

3.1 Performance evaluation of centralized solutions

Before studying state-of-the-art solutions for scalability, let's first analyze how well current implementations perform on one machine. In this section we describe our tests and performance on major implementations for Delaunay triangulations and R-trees.

Virtual worlds demand high responsiveness. To simulate natural human-to-human interaction, occurring events must be propagated timely to the users. Also, any request must be satisfied with the most up-to-date results and in a timely manner, before the state changes make the answer obsolete.

The state of a virtual world is changed mainly by the avatars that populate it. Out of these changes, the majority are position updates [57]. Therefore, the performance of each system depends on the frequency with which these updates occur. Our evaluations focus on the trade-off between frequency of updates and the number of indexed vertices.

We finally claim that a scalable solution must handle this trade-off and provide for any size of the index and any frequency of position updates. Our own contribution, Kiwano and Kwery, will be shown scalable under this requirement –we will show they are capable to dynamically self-adapt to any configuration number of avatars versus frequency of update.

These following measurements anticipate the limits of current virtual worlds and other usages. We analyze the results for some of them.

Additionally, they provide a ground result of the scalability performance of our distributed solutions. As we will see in the next part of the thesis, we are actually able to provide solutions to distribute the computation for Delaunay triangulations and R-trees, with a self-adaptive load balancing. Indeed, in a distributed system a machine cannot exceed these evaluated values. Our next step is to reduce the overhead required by the load balancing procedures of the distributed data structure, such that these limits are closely approached. This subject is covered in Chapter 4 for Delaunay triangulations and in Chapter 5 for R-trees.

3.1.1 Experiments and settings

The data structures of particular interest in virtual worlds are Delaunay triangulations and R-trees, for which we test the limits. We evaluate them as spatial indexes for (1) insertion, (2) spatial queries, and (3) vertex movement. The chosen spatial queries are:

- Nearest neighbor for Delaunay triangulations;
- Axis-aligned range query for R-tree implementations.

To perform our benchmark, we designed one simulator running rounds, each round being made of three steps:

1. **Vertices are inserted:** Vertices are added one by one, that is, the spatial index is incrementally updated;
2. **Spatial queries are performed:** Each spatial query is performed from a new position (nearest neighbor for Delaunay triangulations and axis-aligned range query for R-tree implementations);
3. **Vertices move:** Randomly selected objects are removed and then inserted with the new coordinates, one at a time.

To evaluate the performances, we look at the maximum frequencies supported for these operations versus the number of vertices. The recorded values were:

- The total number of vertices;
- The average time to insert one vertex;
- The average time to move one vertex (as the total time for deletion and successive insertion);
- The average time to perform a spatial query;
- The maximum average frequency supported for position update.

In our experiments a simulation runs for 40 rounds. At each round, 5,000 new vertices are inserted at their respective positions. At each round, the average time per query and per movement is computed for 100,000 spatial queries and 100,000 vertex movements respectively. Additionally, to eliminate the impact of the initial churn when the index is created, we perform 20 additional rounds with 50 vertices inserted at a time.

We performed tests running two geographical distributions for vertex positions.

- **Uniform**. It is the first intuition for avatar placement but, despite that, it is an extreme case. Initially, vertices are placed uniformly at random on the geographical surface. The movement is a teleportation: A fresh random position replaces the current one.
- **Power law**. Both, human [112] and avatar distribution [88, 89] have been shown to follow a geographical power law distribution. In our tests, we generated a set of hotspots geographically distributed using Lévy flights, thus producing densities following a power law distribution. The distance travelled at a position update was also generated using Lévy flights. This resulted in movements that were often within a short range distance and only sometimes further teleportations.

Our benchmark witnessed the same behavior for each of these avatar distributions and mobility patterns. Allow us therefore to present only the results for the uniform distribution.

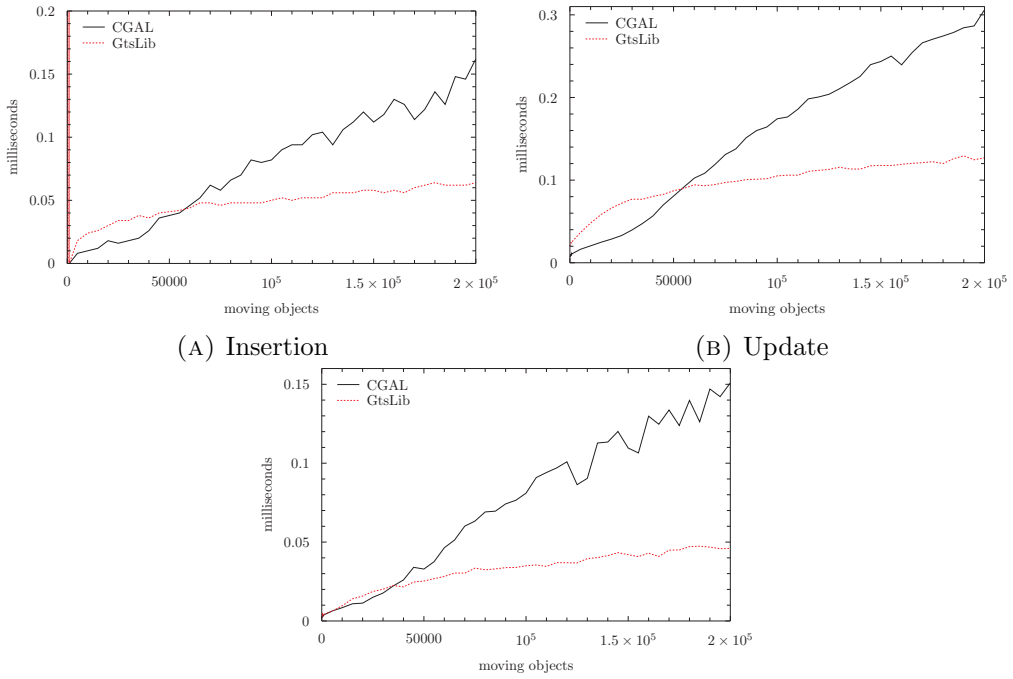
These similar results indicate that we can build a distributed solution that stands any avatar distribution. This result is important and helps us showing that our contribution, Kiwano and Kwery, are general solutions,that are independent of the possible variations in avatar density in time and space.

The simulator is written in Python. The tests have been performed on a computer equipped with an Intel Core 2 Quad CPU Q9400 at 2.66 GHz, running a pre-installed Ubuntu 11.04 operating system. The system had 4 Gbytes RAM and did not use disc caching during the tests. We measured the processor time using `time.clock()`.

3.1.2 Delaunay triangulations

We test the performance using two prevailing implementations, CGAL and GTS. We chose them for their good performance but notably, because they offer the possibility to incrementally update the data structure, and we thought this can lower the cost of maintaining it when dealing with individual position updates.

CGAL [12, 48] (Computational Geometry Algorithms Library) is a popular scientific tool that is actively improving and adding new features. It is light and efficiently implemented. Recent works [54] develop the library with an implementation of Delaunay triangulations on a sphere. This makes it particularly suitable for geographical applications and especially for our intended application on mixed reality. All this, as well as the fast performances described in Figure 3.1 motivated us to chose CGAL to be the Delaunay triangulation for our main contribution, Kiwano.



(A) Insertion (B) Update

(C) Nearest Neighbor Spatial Query

FIGURE 3.1: Performance for implementations of Delaunay triangulation

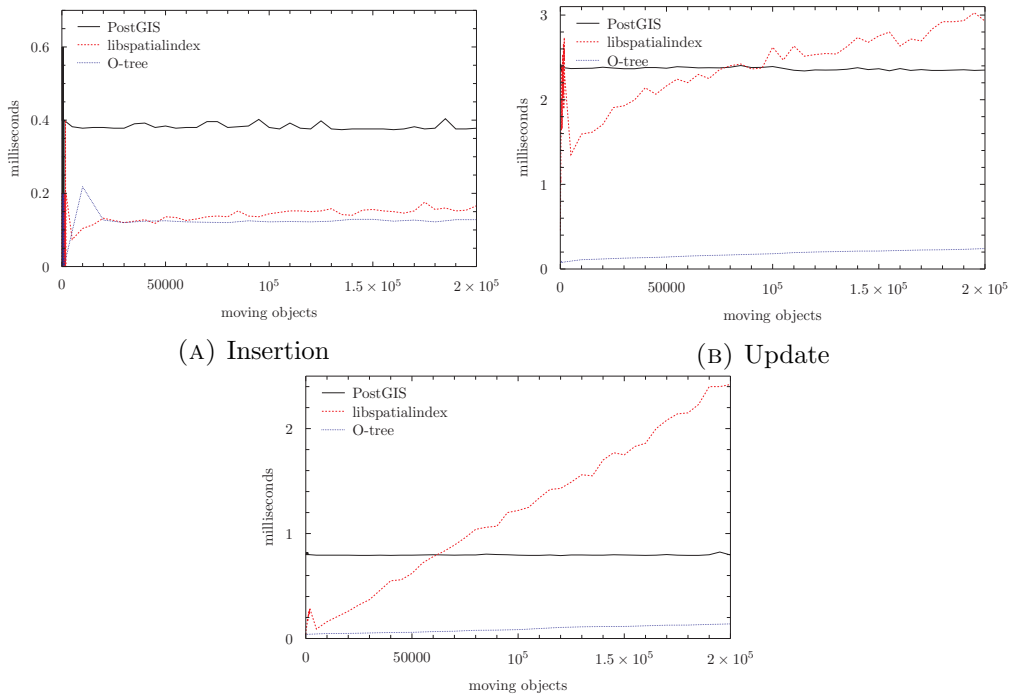
GTS [16] (GNU Triangulated Surface Library) is a Free Software Library intended to provide a set of useful functions to deal with two and three dimensional surfaces meshed with interconnected triangles, employed for computational geometry applications.

As we see in Figure 3.1, GTS is faster for more than 5,000 indexed vertices. Below that, CGAL outperforms GTS. However, for virtual worlds, high update frequencies limit the number of avatars per machine to under 2,000. This is why for our implementation, we prefer CGAL. In what follows we will compare these results with various other use cases.

3.1.3 R-trees

R-trees are one of the most widely used geometric data structures in geographic information systems (GIS) and spatial databases. Their implementations are widespread: From GIS database applications such as PostGIS (the spatial extension of PostgreSQL) [30], Microsoft SQL Server, MySQL, etc., to general purpose libraries such as libspatialindex [24] on unix systems, and publish-subscribe systems. Figure 3.2 compares the benchmark results.

O-tree [106] is a Java implementation of R-trees. It was developed to build several centralized simulators of peer-to-peer overlays in particular to showcase the viability of distributed R-tree based publish-subscribe systems [39, 47, 106].



(A) Insertion

(B) Update

(C) Range Query

FIGURE 3.2: Performance for R-tree implementations

The user defines a multidimensional space to represent objects as axis-aligned rectangles of variable sizes. It provides insert and remove primitives and support axis-aligned range queries. In a centralized context, the tree is stored in RAM. As Figure 3.2 illustrates, this is certainly the fastest implementation for our intended purpose. We employ O-tree as the local spatial index of Kwery, our distributed algorithm for range queries described in Chapter 5.

Libspatialindex [24] has R-tree index implementations that support arbitrary shaped range queries. It has also various parametrization capabilities, including dimensionality, node capacity, memory storage, etc.

PostGIS [30] adds support for geographic objects to the PostgreSQL object-relational database. As well as most GIS, PostGIS also implements the major geodetic systems, its main target being the geographic applications.

The PostGIS database was entirely stored in the RAM memory. Practically, disc accessing time is far more expensive than the RAM, making a huge difference when dealing with frequent data updates. Nowadays server computing power and/or network bandwidth (in the case of distributed systems) become bottlenecks considerably earlier than

the RAM. This is part of the fundamental trade-off and purposes: to provide fast updates and answers to spatial queries, essentially dealing with evanescent data; We need responsiveness but low persistence.

In our approach, by separating the avatar (and object) movement from the rest of the virtual world, we will show how we are able to index the vertices without carrying much information about the avatar (or the respective object). In-memory databases are thus the best storage support for the trade-off. Although widely used for static data processing, PostGIS proves to be too slow when working with dynamic data.

3.1.4 Limits of the data structures

Usually, when designing a multi-user system, the developer is interested in the performance that can be achieved. In our case, the achieved performance reflects directly the trade-off between the number of users and the mean frequency of updates. For instance updating the position in the context of a social network can wait even up to 30 minutes, whereas applications for hybrid reality or first-person shooting games ought to be as frequent as possible, at least 10 updates per second. Based on the above presented benchmarks, we aim at answering the upper limits that are expected for a non exhaustive list of application scenarios, in terms of number of avatars (or dynamic objects) and the frequency of updates.

The space complexity is in both cases linear with respect to N, the number of indexed vertices, but this is not a problem *per se*, because we never approached the limits of machine's RAM. The time complexity is $\mathcal{O}(\sqrt{N})$ when using Delaunay triangulations and $\mathcal{O}(\ln N)$ for R-trees. This complexity however, is not eloquent in practice and does not provide factual limits. And that is because (1) our tests are on a limited number of vertices, for which the shape of the curve may not be noticeable yet and (2) actual implementations rely on a plethora of optimisation techniques which may cause disproportionate curve shapes, as in Figure 2.2(C).

Let's have therefore a closer look on these results and scrutinize them against some application scenarios.

Both solutions enable incremental algorithms which make them suitable for neighborhood updates and spatial queries on moving avatars. Delaunay triangulations, updated incrementally, have a reduced overhead when a vertex is inserted or removed compared to R-tree algorithms. Moving an object in a triangulation will affect a constant number of nodes, and a logarithmic number in an R-tree.

The key characteristic that makes a spatial database a powerful tool is its ability to manipulate spatial data, rather than simply storing and representing it. The join query enables filtering on different criteria involving not only spatial ones. Since R-trees are an extension of B-trees, join queries can efficiently use an R-tree index as a B-tree index. Delaunay triangulation based tools can achieve this at best indirectly, by employing other tools to refine the spatial result.

So when querying with other criteria than only position, R-trees –combined with B-trees– are, if not mandatory, usually far better than Delaunay triangulations.

We represent in Table 3.1 the minimum mean time supported between two successive updates given the total number of mobile objects.

objects	O-Tree	spatialidx	PostGIS	CGAL	GTS
50	0.004	0.026	0.12	0.001	0.001
500	0.04	1	1.19	0.005	0.01
1K	0.08	2.21	2.38	0.01	0.025
5K	0.50	6.76	11.84	0.09	0.17
10K	1.18	16	25.70	0.22	0.46
50K	8.79	108.15	118.60	4.04	4.44
100K	21.00	262	239.20	17.22	10.70

TABLE 3.1: Minimum expected time between two updates in seconds

Given the maximum number of users, this table infers from our previous measurements what is the expected frequency. Conversely, knowing the frequency of position updates, Table 3.2 summarizes the scalability limits in number of moving objects.

T	O-Tree	spatialidx	PostGIS	CGAL	GTS
10ms	150	20	-	900	400
30ms	400	60	10	1000	1000
100ms	2000	120	30	5000	2000
1sec	8000	500	400	26K	17K
10s	75K	7000	4000	75K	100K
20s	110K	12K	8500	100K	200K
1min	300K	30K	25K	650K	1.5M
10min	3M	200K	250K	5M	17M
30min	7M	600K	750K	70M	100M

TABLE 3.2: Maximum number of moving objects for predefined update frequencies (T is the mean time between two position updates)

The results of the benchmark illustrate the scalability limits one can reach with today's systems and they are unsatisfactory for virtual worlds.

At rates of 10 to 30 frames per second, custom systems like CGAL, GTS and O-tree can handle hundreds of avatars evolving simultaneously. However, to build a planetary-scale hybrid reality system, with an unbounded number of users, as depicted in [111],

this is far from enough. Spatial indexes are used to optimize spatial queries since regular indexes do not efficiently handle topology issues such as proximity or containment. Our benchmark shows clearly that performances of spatial databases are way behind dedicated implementations.

In social applications most movements are small [89], withdrawing the infrequent teleportations over long distances reduces the overhead without significant. The frequency of movement can be as low as every 30 seconds. Thus, we estimate the maximum amount of simultaneous users that can be handled with current database solutions in thousands to hundreds of thousands. However, social networks have tens or hundreds of millions of users, exceeding by far these limits.

MOVIES [68] is another centralized solution that uses Hilbert curves to create a new index of the positions several times per second. The latest image of the index is used for answering incoming queries. The proposed implementation supports a load up to 100,000,000 moving objects, 58,000,000 updates per second and 10,000 queries per second, a scenario at a scale unmatched by any previous work. Unfortunately for a spatial index, reducing multiple dimensions to just one cannot totally preserve locality and these techniques produce high rates of inaccurate results, and scalability is still limited by the performance of a single machine.

The interesting feature that makes MOVIES scale is its periodic computation of the spatial index, avoiding simultaneous reads and writes on the same data structure. This is a feature that indeed speeds up the access, and we investigate it in our own implementation of Kiwano, see Section 4.3.

Summary We have seen that for many applications current solutions reach their limits and cannot index the positions of all objects at the required frequency. These limitations arise from algorithms that are centralized and rely on one computer –with a fixed amount of processing power, memory, network bandwidth, etc. So, with certainty, for some number N of objects with a given mean frequency of position updates, the computer will run out of resources. However, it should be possible to scale up to any arbitrary combination by adding enough computers to provide the needed resources.

These results do not take into consideration network latency or other tasks that a machine must accomplish but still, they give us an upper limit on the performance.

3.2 Interest management

Since the early 1970s, when the first multi-user graphic virtual world appeared, the algorithmic complexity of running virtual worlds is $\mathcal{O}(N^2)$, where N is the number of users that are together in the same region. In particular, when one avatar joins the world, all other avatars must be informed. Thus, the algorithmic complexity is $\mathcal{O}(N^2)$. Additionally, for M dynamic –or mutable– objects, the cost of maintaining the world state is $\mathcal{O}(N \times M)$, because, again, all users must receive these updates.

When the number of avatars and objects increases, it becomes infeasible to run the world because (1) each user needs to handle more and more events of the virtual world and (2) the simulator reaches its limits in numbers of avatars and frequency of updates. In this section we discuss how current solutions try to solve (1) by limiting the events that need to be treated by the client. In the next section we will see how distributed data structures address (2) by distributing the load.

To solve (1), virtual worlds assume spatial locality: events have only a local, limited effect. Dually, an avatar is affected by –and thus interested in being informed of– only some of the events happening in the virtual world. That is what constitutes the interest management and the reason why current techniques are space-based. An avatar will receive updates of events happening in some area surrounding its position.

Locality has thus far been defined in terms of space distance. An event is perceived within a circle around its position. As far as we know, locality has not yet been approached based on a neighborhood relation graph.

In this section we recap prevailing interest management techniques intended to limit the load or to overcome, at least for the client, the quadratic complexity. However, in most cases, they fail to preserve linear complexity when the avatar distribution changes.

To view and interact with other avatars, virtual worlds tend to make an anthropological assumption: they must be close in the virtual space.

3.2.1 Region-based publish-subscribe

To allow more avatars to connect, the natural solution was to divide the space into several fixed *zones* or *regions*, each hosted by a machine. Avatars subscribe to the zone they are in. The commercial solutions benefit from this workaround, as it allows all avatars to connect at the same time. However, each zone is limited, and avatars are not together. As we have seen, avatars want to join exactly those zones that are populated [109] and therefore, this does not reduce the load on the client side.

To eliminate borders, an avatar may subscribe to multiple zones. Therefore, borders are not visible any more and the space seems contiguous. The main drawback is that the success of this approach is dependent on how the space is divided into zones. A good data structure and a load balanced distribution are very difficult to construct.

When the virtual world is divided into zones, avatars subscribe to all events in the zone they are located in. But the world is not contiguous: two zones are two separate virtual worlds. As we will present in Section 3.3.2, this does not scale due to the incapacity to balance the load between servers.

The region-based interest management works well when avatar and object distribution does not change over time. But this is not the case either in real life, or in virtual worlds. Real-life activities will increase the number of people on highways on rush hours and in tourist hotspots on holidays. MMOGs [99, 100] and virtual worlds [109] have a peak of users around 8PM. Their hotspots evolve according to the activities taking place.

When regions are small enough not to saturate a machine, the area-of-interest may either be the entire region [114] or a smaller area within the same region. It may also span over multiple regions at once [50]. A problem is that regions are not sufficiently fine-grained. Large groups of avatars will appear and disappear when changing a zone. That's why it has often been combined with aura-nimbus. The intersection with an avatar's area of interest will be performed only on the nearby zones.

3.2.2 Aura-nimbus

The *aura* is the space around an avatar's position that bounds the effect of its events. Dually, the *nimbus* is the area containing the relevant events for an avatar. He can perceive other avatars, objects and events located inside. The nimbus has often been referred to as *area-of-interest* [85], awareness area [83, 84], domain of interest [95] or aura [50].

Avatar distribution in virtual worlds is not uniform but follows a power law distribution. In fact, this resembles the human distribution [88]. This makes an area of fixed size too large to handle in crowded places because it provides too many events. At the same time it is too small to socialize in desert areas because there is no hint where others are.

Other solutions [63, 80] propose that each node to individually determine the radius of their areas of interest according to their capacities –with an emphasis on bandwidth. But this leads to inconsistencies due to the asymmetry of the relation. In VELVET [63] Alice may see Bob while Bob may not see Alice when the area of Alice is larger and includes Bob. So, if Alice kisses Bob, he will never know it. In VON and VAST [80]

Bob will not close the connection with Alice, while Alice still sees him, but this may overcrowd his area.

3.2.3 Summary

Studies have shown that a user's interest is focused only on a very limited set of objects. With this in mind, using interest management is really important in order to reduce the overload for the client. Current research is still fervent in trying to provide realistic environments with a minimum set of transmitted events.

Donnybrook [45] proposes a fixed size interest set, the set of other avatars to whom the player is paying attention. Players receive frequent updates only for avatars in the interest set, while for all others, just one update per second. Scalability is improved up to 900 simultaneous players with low impact on user experience. The drawback is that it requires algorithms on the client to estimate attention. Predictions are specific to shooter games and not always accurate.

Locality of events is well exhibited by the aura-nimbus area of interest. But, we have seen that this approach is not suitable when the density is skewed due to the power law avatar distribution. We will see in the next section how actual solutions vary the size of the area of interest in order to avoid having too many avatars and events for some and the feeling of an empty world for others.

Region-based approaches are less costly than computing the intersection with areas of interest. But fixed zones fail when the avatar distribution changes unpredictably over time. For dynamic zone allocation, it is hard to determine the appropriate size of a region [71]. With a self-adaptive granularity, dynamic regions seem promising, but as we will see in the remainder of this chapter, current implementations are not sufficiently general and efficient.

So far, all existing interest management techniques have been space dependent. They are consisted of all avatars and objects located within some surrounding area. To our knowledge, no interest management technique has split the three world components: static terrain, mutable objects and avatars. Also, no interest management provides information using a graph of relationship.

Our contribution addresses these issues. In Part II of the thesis we propose to separate the interest for each of these components and provide solutions to handle each of them. The terrain is static, usually very simple, and does not raise a scalability problem as it can possibly be maintained by each user. To see and interact with the surrounding avatars, we propose Kiwano with a neighborhood relation defined over the graph of

proximity relations (Delaunay in our case). Kiwano provides a constant number of neighbors in average independently of the avatar distribution and mobility pattern. Also, to retrieve the relevant dynamic objects, we propose Kwery, a distributed index for publish-subscribe.

3.3 Data distribution

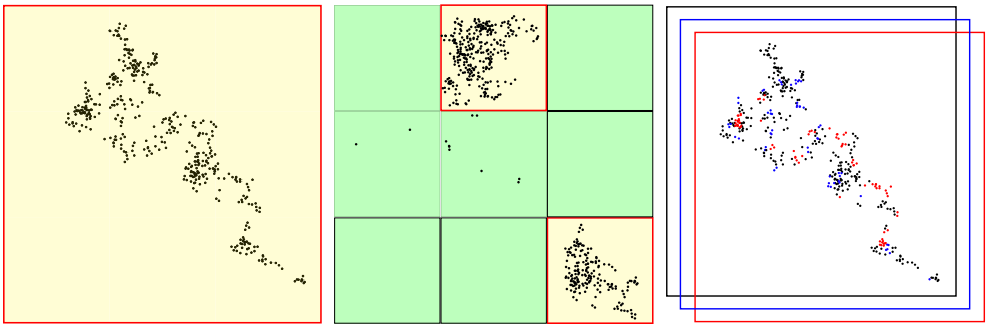
The aforementioned interest management techniques may alleviate in many cases the load handled by the clients and the traffic but does not reduce the computation that needs to be supported by the server. The simulator still needs to compute who needs to be informed by an occurring event. Therefore, interest management alone is not sufficient to provide scalability.

Solutions to enable scalability for virtual worlds must pay attention to the cost of accessing data. The abstraction chosen for data structure is crucial and must reflect the desired trade-off between the properties –scalability, consistency, or responsiveness for the data [51]. For example, to retrieve the neighborhood, the appropriate data structure is the Delaunay triangulation rather than R-trees, which may be preferred to perform spatial containment queries.

The following remark is important. A scalable architecture must be built upon a distributable data structure. A complete graph where data is propagated and accessed via the edges is not distributable since each operation will access all nodes. On the other hand, a balanced hierarchical tree structure where, at each step, queries and updates are propagated only to the concerned descendants, can be distributed.

Limits of the chosen data structure may reflect in the features –or the lack thereof– of the game logic. For instance avatar visibility may be limited in order to allow client machines to maintain an up-to-date image of the world state and to limit the number of events needed to be treated. Avatar capabilities may be reduced in order to reduce the number of generated events –*e.g.*, the access to a restricted set of objects in the immediate proximity. According to the nature of the world, these constraints may correspond to certain game scenarios.

Because events have a limited effect and avatars ,limited capabilities, in virtual worlds data is accessed by proximity. This is why most solutions assume objects and avatars are strongly dependent on the geographical space. With this assumption, solutions will focus on a spatial distribution of the load, while aiming for a balanced distribution of the total load.



(A) One zone (B) Zones (C) Shards

FIGURE 3.3: Avatar distribution for data partitioning

3.3.1 Shards

An early solution for scalability were the *shards*, introduced by Ultima Online [34] game, where the approach is to divide the set of avatars, instead of the geographical space. Their solution is to have more copies of the same world, also called shards or *instances* [77], each hosting just a subset of avatars. Shards are limited by the capabilities of the server machine in the number of concurrent users and multiple copies are deployed as users continue to connect. Shards do not communicate, thus users on one instance cannot interact with users on another instance. This workaround is only useful for games or services that can integrate these constraints of limited and anonymous interaction.

Of course, we are able to run as many instances as we need and each instance will handle exactly its capacity, but users are no longer in the same virtual world. If Alice and Bob have a rendezvous at the town hall, they will not meet each other unless they are in the same shard. They are not actually in the same world.

World of Warcraft [37] used this solution at the beginning. Nowadays they will migrate your character to the realm of your choice for only 20€. Also, they offer to migrate your avatar from selected high population realms to low population realms for free. In that respect it seems they're not trying to do business with the migrations, but instead discourage people from overcrowding the most popular realms by migrating there.

Shards seem to be a solution inspired by poker games: When more users want to play, just deploy more game servers. All in all, they do not scale virtual worlds.

3.3.2 Zones

To create the illusion of a contiguous space, it seems natural to partition the territory in contiguous non-overlapping *zones* [77], each limited in the number of avatars to less than

the maximum manageable by one server. This means that zone servers –also called *sims*– receive the position updates of the objects within their zone and inform the avatars of the modifications occurring in their proximity. In the literature they also appear under the name of *islands*, *regions* [29, 32], or *cells* [49], reflecting the various specificities.

Islands In their simplest flavor, zones do not communicate. This approach creates completely disconnected worlds, integrated into the gameplay as islands. Each one has borders and to travel between them users need to disconnect from the current and reconnect to the target zone.

As people want to socialize and concentrate where the crowds are, many zones remain empty while a few become overcrowded. Moreover, these islands are disconnected so avatars want to join exactly those islands that are populated. Since exactly the most attractive zones are saturated, this approach does not overcome the scalability issues.

Regions To simulate a contiguous space, some solutions split a larger territory into fixed regions with invisible frontiers. Zone servers take care of the avatars and objects located within their coverage. Updates and object inserts are sent only to the zones where they occur.

This approach is very efficient when object distribution remains the same over time so each zone is in charge of a roughly constant number of objects.

However, in actual systems, *e.g.*, Second Life, most of the regions are empty, 30% are never visited in a six day period, while only 1% are overloaded [109, 114]. The avatar natural distribution follows a power law curve but since crossing borders necessitates complicated trade-off protocols and is not seamless for the avatar, the density becomes even more skewed (see Figure 3.3).

As speaking of avatars and moving objects, it is not senseless to expect them to move *en masse* to one place and thus change dramatically the density in a given zone. To take this into account, some zone servers have to be underloaded most of the time, just waiting for the once-a-week or once-a-month event that will saturate them. This is highly inefficient as the resources allocated for the empty regions remain unused. Even so, the best provisioned servers can support only a limited number of users, such that flash crowds, and flocking are not possible. Large groups are still impossible. For instance, in Second Life meetings gather less than a hundred simultaneous avatars.

Dynamic space partitioning Their lack of flexibility and the limitations imposed on the mobility patterns make the fixed zone approach overly costly and unusable. For

that, some solutions adapt the allocation of resources according to avatar distribution and density fluctuations [49, 65, 66].

However, the name is unfortunate and misleading. The space is partitioned *a priori* into *tiles*, each small enough so as not to saturate a machine whatever the density. A computer node of the system may take care of one or several at a time, dynamically (re)distributing these tiles on the processing nodes. Using Delaunay triangulations of fixed points in space, [66] proposes an interest management technique, which takes into consideration obstacles in the virtual world. A balanced load has been achieved but experiments have been conducted only with uniform distributions of avatars and obstacles.

This approach is still space based and there is a high cost of serializing and transferring entire zones –with objects, avatars, etc.– from one node to another to restore the interest management. These events must occur only occasionally. In conclusion, it is hard to determine the appropriate zone granularity. Pikko Server [38] uses this technique but transfers only the management between the cores of the same machine.

Dynamic space partitioning is a great improvement from static zones and shards. However, they are not prepared for a highly dynamic distribution of avatars and events. Scalability is geographic dependent, avatars and objects are bound to the zone they are located in.

3.3.3 Tree overlays for range queries and publish-subscribe

Range queries and publish-subscribe are largely interreducible [44]. Updates correspond to publications. The difference between queries and subscriptions is that queries are discarded as soon as they are answered while subscriptions mean that all future notifications will be received until the subscription is cancelled. To resolve subscriptions with an index –distributed or not– nodes must keep the incoming queries and check if new updates match any subscription. This procedure can be launched at every new update or periodically, for the accumulated updates. In Chapter 5.6.2 we describe how we implement a publish-subscribe system for Kwery, a hierarchical distributed index with zones that reshape dynamically in order to follow the distribution of events.

Range queries Spatial indexes are used to optimize spatial queries since regular indexes do not efficiently handle topology issues such as proximity or containment. Data distribution commonly uses R-trees [78] or Quadtrees [72]. Many of the following examples were combined with peer-to-peer overlays to build scalable distributed data structures [76]. They were intended to provide:

- No central node is used for data addressing;
- Nodes are dynamically added to the system when needed and data is dynamically (re)distributed;
- For that reason, a query may land on the incorrect node and then it will be forwarded to the correct one which will respond and update the user information.

Let's survey a few solutions.

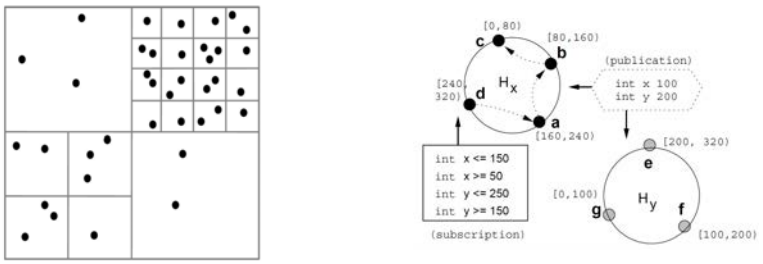
R-trees have been adapted to fit different performance requirements. R*-trees [42] speed-up query answers by maintaining both, minimal coverage and overlap. R+-trees [102] completely avoid the overlap. Therefore, an access follows a single path. However, both R*-trees and R+-trees require additional verifications for overlapping and balancing techniques for insertions and updates, which make them costly and inconvenient for highly dynamic data.

Berchtold et al. propose X-trees [43] to avoid overlap of bounding rectangles by using an optimized organization for high dimensional spaces. To reduce the overlap of R*-trees they use supernodes which may become larger, and a new split algorithm. However, supernodes require more resources and their areas are prone to too much increase.

Quadtrees [72] are used to partition spatially located information in a two-dimensional space. The entire space is covered and when the load of a region becomes too large it is split into four contiguous, non-overlapping regions of equal size, as illustrated in Figure 3.4(A). However, there are no current viable solutions to merge two regions. The resulting tree is not balanced and churn may create long paths.

Several tree-based indexes have been proposed to optimize nearest neighbor queries. For virtual worlds and in general for metric spaces, rectangles may not enclose the nearest objects. SS-trees [113], SR-trees [69], and M-trees [59] employ bounding spheres. Their main drawback is the resulting large overlap.

Publish-subscribe Mercury [44] provides a scalable protocol for multi-dimensional range queries on top of which implements a publish-subscribe system, see Figure 3.4(B). It partitions the data across k rings, where k is the number of dimensions of the representation. Each node of a ring is responsible of a range in that specific ring and maintains the keys in its contiguous range and $k-1$ random sample links to nodes belonging to other rings. To deal with a possibly skewed key space, Mercury explicitly balances the load among the nodes of a ring.



(A) Quad tree [96] (B) Mercury [44]

FIGURE 3.4: Spatial indexes for publish-subscribe

SPEX [96] is a recent architecture aiming at providing a cloud-based solution for publish-subscribe in virtual worlds. To distribute the load, it uses dynamic Quadtrees of the region space. This solution scales up by providing fast distribution when the load increases rapidly but it doesn't scale down. Merging under utilized Quadtree cells is not feasible with an avatar distribution in power law, see Figure 3.4(A).

Although our presentation is limited to a few prevalent examples, the limitations extend to all tree structures.

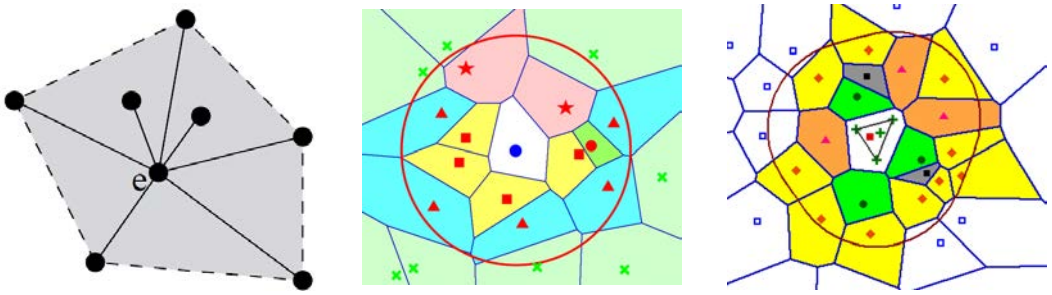
3.3.4 Delaunay triangulations

An important structure for data distribution are Delaunay graphs –or the dual Voronoi diagrams– because they exhibit locality properties and full connectivity. The important feature is that the global data structure can be correctly computed and constantly updated in a distributed manner among the nodes of the system.

They have been widely used for data distribution and dissemination in peer-to-peer systems in general because connecting nearby peers improves latency. In this work we will focus on state-of-the-art virtual worlds.

They basically provide an overlay graph to distribute and disseminate the data among users nearby in the virtual world. They have been successfully used by several peer-to-peer virtual worlds [41, 52, 80, 83, 84], some examples illustrated in Figure 3.5. In these systems, each node maintains a local triangulation of its position and the neighbors' positions. They are connected and exchange local information as they move in the virtual world.

In Solipsis [84] each node calculates a convex hull of its neighbors in the triangulation and queries those inside to inquire if they have detected changes. To dynamically compute the structure, in VON/VAST [80] each node permanently transmits to their neighbors information about the visible avatars and objects. From the received information, nodes extract only what is of interest and simply discard the unnecessary.



(A) Solipsis [84] (B) VON [80] (C) VoroGame [52]

FIGURE 3.5: Neighbors in peer-to-peer distribution with Delaunay graphs

When avatars move, they connect from hop to hop to the new neighboring nodes, which is convenient given the connectivity of the Delaunay graph. In virtual worlds, movements are mostly small and for these there isn't much communication overhead to maintain the overlay. However frequent long distance travel may lead to disconnection.

Delaunay triangulations are not suitable for three dimensional spaces [114]. The evoked drawbacks of the current solutions relate to the interest management or to the general problems of peer-to-peer systems.

In conclusion, we consider Delaunay triangulations promising for a scalable and dynamic data distribution. Kiwano, our contribution, employs them for modelling the distributed neighborhood relation between avatars.

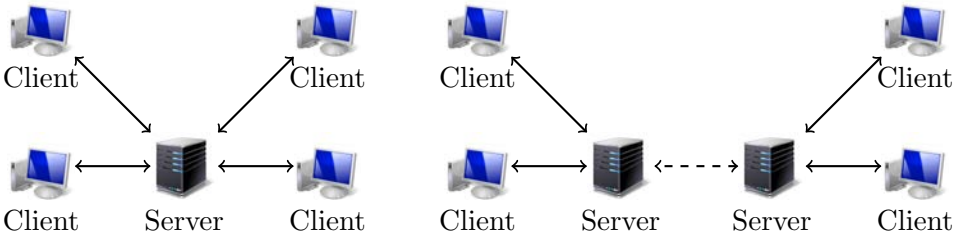
3.3.5 Summary

All these models for data distribution have a space based approach. For this reason they fail to provide an adequate load balancing, independently of avatar distribution and mobility. We consider that an important step to take is to isolate avatar management from the virtual world because this is what generates an imbalanced load. At the same time it is important to provide an adequate interest management technique. It ought at least to be symmetric, and to give a roughly constant number of relevant neighbors independently of the density.

3.4 Architectures

The state of the art for virtual worlds addresses the scalability problem in three ways:

- Trying to augment the number of avatars per machine. This implies reducing the cost per avatar. These are usually simple client-server architectures, *e.g.*, Minecraft, but also more complex, involving multiple servers, *e.g.*, Second Life.
- Increasing the allocated resources. Distributed, multi-server solutions rely on this by fragmenting the space and limiting avatar visibility.
- Using the resources added by each user joining the world. Peer-to-peer architectures prove high scalability and low maintenance cost because they assume that each user brings his own resources.



(A) Client-server (B) Multi server

FIGURE 3.6: Server based architectures

3.4.1 Server based

In the typical client-server architecture, the server maintains the master global state of the world. It receives updates for the mutable objects and avatars, resolves conflicts, computes the new state and updates the copy states of the concerned avatars.

All commercially successful virtual worlds are built upon a server-based architecture. From the simple ones where servers can be deployed independently by the users, *e.g.*, most first person shooter games, Minecraft [32], to elaborated architectures with communicating nodes and relying on various remote services, such as Second Life or World of Warcraft [37], their performance relies on a limited number of server nodes.

Even the multi-server solutions rely on a space division. The cost of the computation is bound to be quadratic, and depending on how complex the virtual environment is, virtual worlds report different upper limits. Less than 100 users/region in Second Life [32], a few thousands for Minecraft [DKV13], around 120 users per shard for World of Warcraft. Eve Online, which recently reported 3,000 users, relies on particular features that allow time dilatation. This is not applicable in all scenarios [79].

Pikko Server [38] have developed their own optimized dynamic space partitioning algorithms that permanently re-evaluates the boundaries. It transfers players dynamically

between game servers to avoid overload. It was used in early 2012 to break the world record for concurrent users with thousands of avatars in a first person shooter battle. Pikko Server is an incentive to distributed, self-adaptive architectures. However, the cost of transferring chunks of the virtual world is high. So far it has been tested on one –a powerful 8-core– machine only.

Due to a plethora of worlds, new solutions are usually narrowly targeted. Bowman et al. [90] separate the treatment of the virtual world into discrete components, such as client management, physical simulation, script processing, and scene persistence. Each component is run as an independent service, and the components are connected through the shared "scene graph" –the state of the virtual space that they collectively simulate. They reported support for up to 1,000 simultaneous avatars in OpenSim [87] for a slightly reduced but acceptable frequency updates. Of course, the machines running each service eventually run out of resources, so the system does not actually scale.

However, the separation of concerns, *i.e.*, the scene from the actors, is a promising technique because it allows geographic independence for objects and avatars. Moreover, it allows the scalability problem to be solved differently for each, avatars and objects, being aware of their different requirements.

We will discuss Minecraft and Second Life mostly using OpenSimulator [29], architectures in detail in Chapters 6 and 7, respectively. These popular virtual worlds are two foundational case studies for our scalability solution, Kiwano.

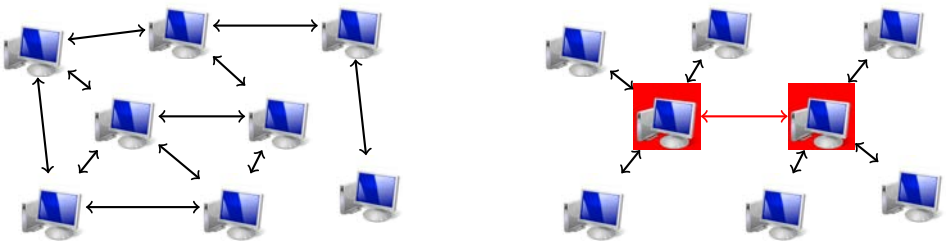
3.4.2 Peer-to-peer

A virtual world requires more resources for each client joining the world. But each client also comes with his own resources: each one uses a computer connected to the network. Peer-to-peer architectures exploit this feature by passing to the client computation usually preformed on the server side. A peer-to-peer system is a communication graph at the application layer. It is incrementally maintained by the peers.

The first peer-to-peer virtual world is Solipsis [83, 84] proposed by Keller and Simon. VON/VAST [80, 81] and Voronet [41] use similar approaches based on Delaunay triangulations. The overlay connects those that are prone to exchange information, close in the virtual worlds. So, the latency is directly reduced.

VoroGame [52] deploys two layers of peer-to-peer overlays: a DHT[1] to maintain game objects and a Voronoi based overlay for neighbor discovery (see Section 3.3). While these

[1] Distributed Hash Table



(A) Peer-to-peer (B) Hybrid

FIGURE 3.7: Peer architectures

approaches benefit from the advantages of both architectures, one major drawback is the cost of maintaining two separate overlays.

Walkad [108] also addresses scalability for the entire scene –objects and avatars altogether– and was used to prototype a peer-to-peer Second Life [107]. The object management is distributed among the peers using Kad and each one accesses its surrounding, fixed-size area of interest.

Colyseus [46] is the infrastructure built on top of Mercury [44] for implementing a scalable Quake II. Each is assigned a contiguous region of the virtual world in the form of an axis-aligned rectangle. When the load becomes imbalanced, load balancing is performed by adjusting peers' regions with their neighbors.

Coupled with peer-to-peer overlays, DR-trees [47] have been designed for scalable dissemination of information, taking advantage of the logarithmic complexity of hierarchical structures. They adapt various variants of R-trees to apply spatial filters.

VBI-Tree (Virtual Binary Index) [82] is a peer-to-peer framework to build multidimensional indexes based on balanced trees such as R-trees [78], SS-trees [113], M-trees [59], etc. Each peer maintains an internal node and a leaf. But, proximity is not reflected and the number of hops for transmitting information is increased (about 10).

Despite the amount of research carried on, these solutions have failed to be adopted by the industry. To explain this, many reasons are invoked. Particular to games is the problem of cheating [114]. But this does not fully explain the phenomenon because Minecraft [28] attracts players because they are able to tinker the game.

Drawbacks of peer-to-peer architectures are the communication overhead [71] and the complex load balancing [93]. One of the most prominent reasons is the weak performances of these systems which are degraded by the capacities of the slowest peers. To maintain the Delaunay triangulation, the Voronoi diagram or the tree hierarchy, each

peer needs to permanently update its structure [71]. This discards peer-to-peer solutions for lightweight devices running on battery and with a poor wireless connection like smartphones and wearables.

3.4.3 Hybrid

Peer-to-peer systems assume homogeneous users that enter the virtual world using a powerful well connected computer. Some propose to exploit the inherent heterogeneity of clients and to use exactly the best equipped nodes to maintain and deliver the world state [115]. They appear under various names, *superpeers* [115], *hubs*, *coordinators* [85].

One of them is MOPAR [116]. Similar to VoroGame [52], it uses a two layered overlay for game objects and neighbor avatars, but the computation is done only on the superpeers. It uses fixed regions.

Hierarchical peer-to-peer systems have also been proposed [73]. Peers are organized in disjoint groups coordinated by superpeers that receive and forward further messages.

Superpeers have higher demands for resources. However, they are selected among clients, so they are an important point of failure [71].

3.4.4 Towards cloud infrastructures

Cloud infrastructures do provide homogeneous, powerful and well connected computers. Computations should, when practical, be performed in the *cloud*, *i.e.*, data centers. They provide some clear advantages.

- Many computers are available at any time and can join the system on-the-fly.
- They are clearly more reliable than clients' machines when compared to peers.
- Automating this process reduces costs and human errors.
- The developer keeps the control, namely, he can choose any configuration and can anticipate the final system performance.
- The latency and throughput can improve significantly over a peer-to-peer solution.
- To reduce the latency, servers can be physically placed close to the users' location. They may be distributed worldwide.
- Data centers become, at the same time, more powerful, cheaper, environmentally-friendly, and more accessible.

Commercial solutions use server farms to host regions or shards, or to provide some services, such as login, item storage, accounts [28, 29, 32, 37].

All the aforementioned features allow massive scalability, self-adaptive, distributed systems. Research and industry look towards cloud-based solutions yet, the existing applications are still limited.

These emerging systems can greatly benefit from the existing research. Machines in the cloud can communicate in a peer-to-peer fashion, without being concerned where other machines are located. Of course, it is not feasible to port peer-to-peer algorithms to the cloud directly, because a blade in a data center is much more powerful and viable than a random peer. Plus, treating peers individually results in a huge overhead.

It should be possible, for instance, to deploy the superpeers of a hybrid infrastructure in the cloud. However, as the scalability of these systems is limited, they can be improved. For that matter, a promising research direction is the use of peer-to-peer overlays over the cloud.

This means that with an adequate approach, various peer-to-peer architectures where peers run on blades in data centers should be implementable. Using the peer-to-peer communication overlay, scalability in the cloud should be unstoppable.

3.5 Summary

We have seen why building a virtual world is a difficult task. Of course, one machine eventually runs out of resources, and therefore we need a distributed system. But even with enough resources, the scalability depends on how well the distribution is performed.

First of all, a system cannot scale faster than linearly with respect to the centralized solutions. In other words, two machines cannot host more avatars than the double for one machine. In this regard, we assessed the limits for the main indexing structures.

Then, the architecture must provide mechanisms to cope with variations in density and to be ready to allocate resources dynamically where they are needed. As we have seen, current architectures are not capable to perform a self-adaptive load balancing. And this is a key feature to be able to take advantage of the resources at their fullest.

The reason is mainly because the chosen data models are not actually distributable. They all rely on spatial division. Even for those that are said to balance the load dynamically, they hand over complete fragments of the virtual world, and this is very costly.

Additionally, to allow the client to run, he must be informed only of the significant events nearby, because his machine is limited too.

To conclude, we remind you that there are many available resources: data-centers with a myriad of computers. There are social services with millions of users and games with hundreds of thousands of subscribers. Yet, virtual worlds remained limited to just a few hundred avatars in the same contiguous space. The problem, we thought, must be redefined from scratch. We realized that handling the avatars and mobile objects, and handling the décor are two different problems. They should be addressed separately.

Part II

Contribution: Scalability for Virtual Worlds

Chapter 4

Kiwano: Avatar Scalability and Neighborhood Updates

Contents

The main contribution of this thesis is Kiwano [DK15], a distributed system that enables an unlimited number of avatars to be and interact in the same virtual space. By separating the virtual world components –avatars, moving objects from the static décor– we take a novel approach and introduce a neighborhood relation between avatars. In

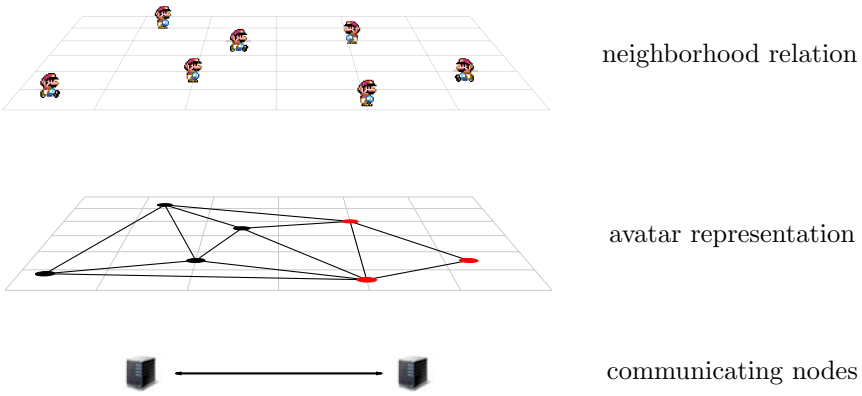


FIGURE 4.1: Different overlays in Kiwano. Neighboring avatars, vertices, and nodes

Kiwano we employ the Delaunay triangulation to provide each avatar with a constant number of neighbors independently of their density or distribution. The avatar-to-avatar interactions and related computations are then bounded, allowing the system to scale.

In Kiwano, each machine hosting the world (or node) takes in charge a set of avatars based on their geographical proximity. Nodes constantly exchange avatars and adapt in order to maintain a balanced load.

The optimal number of avatars per CPU and the performances of our system have been evaluated simulating tens of thousands of avatars connecting to a Kiwano instance running across several data centers. These results exceed by orders of magnitude the performances of current state-of-the-art implementations.

We introduce the notion of neighborhood in Section 4.1 and motivate our choice for Delaunay triangulations in Section 4.2. Kiwano distributed algorithms are described in Section 4.3 and how they are used to make a virtual world scalable is explained in Section 4.4. In Section 4.4.2 we give some details on the actual implementation. The simulations and performance measurements of Kiwano are shown in Section 4.5.

4.1 Avatar interest management

When separating the management of the three components –avatars, objects, and terrain– the notion of area of interest changes. Although they must be located in the proximity of the concerned avatar, the way we pay attention to them differs. The interest in avatars is for social interaction, objects may be tools, and the décor is a setting. This is the reason we employ a different notion, of neighboring avatars, disconnected from the terrain.

In Kiwano, we take a novel approach: users can solely see and interact with their *neighbors*. Kiwano's *neighborhood* relation is designed such that the number of neighbors

remains within a given range regardless of the avatar density distribution. The neighborhood relation is also the base for defining the distributed data structure in Kiwano. In our approach, the algorithmic costs are distributed as follows:

- The client has a bounded number of neighbors. Thanks to this result, building the graphical representation of other users' avatars and compute the avatar-to-avatar interactions are constant. These computations and the associated bandwidth do not grow with the number of avatars. The scalability is attained by allocating enough processing power and bandwidth to each user.
- With Kiwano, an unlimited number of avatars can now enter the same region. The neighborhood is recalculated as avatars join, leave, or move. This is done by Kiwano in a distributed fashion with a linear global cost with respect to the number of avatars. Adding more processors to the system is the solution to handle more users.

With this approach, the number of avatars in a contiguous space can grow to arbitrarily large numbers with a constant computational cost per avatar for the client –bounded interest management– and for Kiwano –linear complexity for maintaining the world.

4.1.1 Neighborhood relation

The notion of *neighbor* evokes some sort of proximity. In this section, we define the notion of *neighborhood relation* for virtual worlds: what it means for two avatars Alice and Bob to be neighbors and how to compute an avatar's neighborhood.

We have seen in Chapter 3 that actual notions of interest management define, under various names, an *area* of interest. Interaction between avatars is possible if they are in each other's area, hence the geographical dependence.

We abstract this notion of interest between avatars by considering the graph of possible interactions where avatar positions constitute the vertices. Alice and Bob will be connected in this graph if they can see and interact with each other, that is, if they are in each other's area of interest. The limitation of the zone based approaches is that the avatar distribution is skewed. Some avatars will see too many neighbors while others not enough, they may even be disconnected.

In our approach we consider an avatar's neighborhood to be a subset of surrounding avatars of roughly fixed size, relevant for the social activities that take place in virtual worlds.

The main goal of connecting avatars only to their neighbors is to reduce the number of avatar-to-avatar interactions. Indeed, each interaction comes with network and computational costs at least proportional to the number of neighbors:

- The user's terminal has to render the graphical representations of others' avatars.
- Collisions and other virtual physical interactions between avatars have to be computed.
- The dynamic data of other avatars (*e.g.*, positions, body postures, state changes) have to be updated through the network.

Hence, to ensure scalability, we need to bound the number of neighbors. At the same time, the neighborhood relation should be meaningful for social interaction. It should satisfy the following constraints:

i) The number of neighbors should be small enough to ensure practical scalability in the aforementioned setting. This is a mandatory condition.

ii) The relation should be symmetric, so no avatar can see another avatar without being seen.

iii) The nearest avatars should obviously count as neighbors.

iv) A similar property with three avatars: If they are alone inside a bubble they should all be neighbors.

v) The number of neighbors should not be too low, even when the density is locally low, such that some online social activity is still possible.

We have seen in the previous chapter that the most common interest management technique, the set of avatars within a distance, the area of interest, failed to satisfy these conditions. Because the density of avatars may vary sharply, the range of the neighborhood needs to be adjusted for each avatar independently. In this way, the number of neighbors is bounded and condition (i) is satisfied. Then, while conditions (iii) and (v) are also satisfied, the remaining are not.

Condition (ii) avoids invisibility. It also ensures that interaction will be perceived by all those involved. When an avatar sees another they are both able to interact. Bob will see Alice whenever she kisses him. Conditions (iii) and (v) circumvent the limitations imposed by a fluctuating avatar density. In an open space one will see where are the closest neighbors, while in a crowd one will pay attention to those in the immediate

proximity. One will never be alone. These conditions indirectly imply a degree of geographical independence for the avatars.

To distribute the load and, at the same time, to easily gather proximity information, some solutions in virtual worlds make use of the Delaunay triangulation of the avatars' positions [41, 80, 83, 84]. Certainly, as presented in the background, Chapter 2, Delaunay triangulations and their dual, Voronoi tessellations, have been widely employed as data structures for proximity information.

We observe that a relation defined on a Delaunay graph satisfies all conditions. Arguably, Delaunay triangulations face two issues. The first one is that the properties of the Delaunay triangulation in three (and higher) dimensions are not as good as in 2D. For instance, for N vertices the number of edges in the 2D graph is $\mathcal{O}(N)$, while in 3D is $\mathcal{O}(N^2)$ in worst case. Nonetheless, most virtual worlds take an anthropological assumption. As humans usually do not fly, most of them place avatars on a surface, sometimes with small variations. On Earth, unless an aeroplane is involved, longitude/latitude is a good indicator of position. Therefore, using two dimensional coordinates to compute proximity will suffice for most use cases. The other issue is that the average of the numbers of neighbors in any Delaunay triangulation is 6, a number too low for most social activities. Condition (v) may not be satisfied. This has to be improved but gave us a hint on how to define the neighborhood relation in Kiwano, and finally even more: a whole series of relations suiting different needs. We detail this in what follows.

4.1.2 K^{th} power of Delaunay graphs

We note D or D^1 the Delaunay triangulation graph of the avatar positions and D^k, the k^{th} power graph of D. For simplicity we employ the same notation D^k for the corresponding adjacency relation. Thus, for a relation D^k the neighborhood of a vertex v is $D^k(v)$, the set of adjacent vertices.

As a reminder, v and u are said to be neighbors in the graph power D^k, if their distance in D –*i.e.*, the minimum number of hops between them– is at most k. Or, in other words:

$$D^{k+1}(v) = D^k(v) \ \cup \ D^1(D^k(v))$$

The first three levels of this relation are represented in Figure 4.2. To improve readability, the dual Voronoi tessellation is depicted. The color is lighter as the level increases.

The number of neighbors of an avatar v, *i.e.*, the size or cardinality of $D^k(v)$, noted $|D^k(v)|$, is not constant and depends on the distribution of other avatars in the proximity.

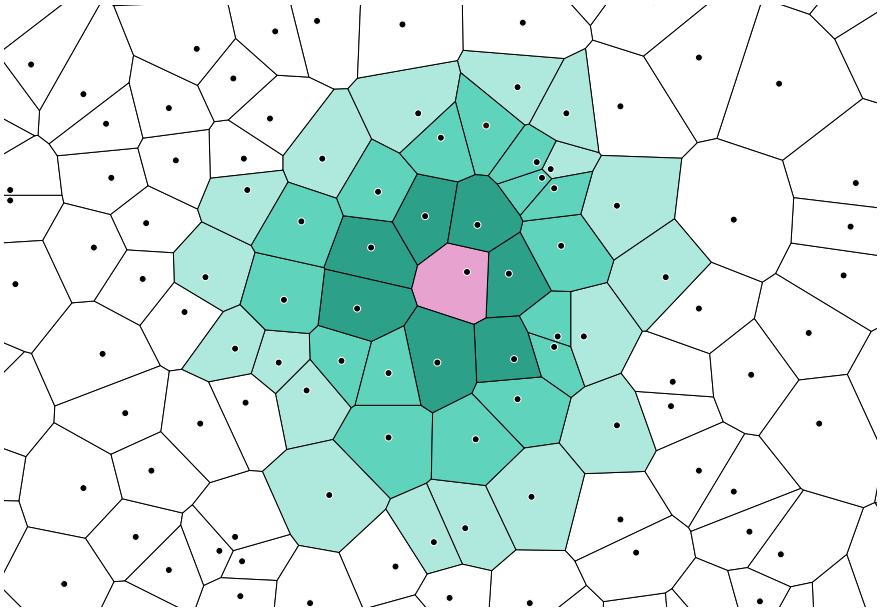


FIGURE 4.2: The first three levels of an avatar's neighborhood

To determine the range in which the number of neighbors evolves, we simulated avatar positions with various distributions: uniform, Gaussian, and power law, with varying parameters. A density distribution of avatars in power law is considered realistic by many studies (see Chapter 2 and Section 3.1, also [88, 89, 109, 112]).

We summarize the outcome of the simulations for the power law distribution for D^1 to D^6 in Figure 4.3. We trace the percentage of avatars v with a given size of $D^k(v)$. Due to the local nature of the Delaunay triangulation, these percentages are stable, in the range of a Gaussian bell distribution, regardless of the total number of avatars. Actually, measurements with uniform and Gaussian densities produce very similar results. Only configurations built carefully and on purpose, with perfectly aligned positions, have been able to grow the neighborhood to large numbers.

Therefore, given the nearly null probability of the alternative, we can safely consider $|D^k(v)|$ to be bounded. Hence, the D^k relations are good candidates to be the neighborhood relation chosen for the first implementation of Kiwano.

Let's now revise our previously self-imposed conditions:

(i): The number of neighbors is bounded to low enough numbers, see Figure 4.3.

(ii): D^k is an undirected graph and thus provides a symmetric relation.

(iii): The k-nearest avatars are guaranteed to be neighbors in D^k and usually the number of nearest neighbors is greater, as a result of the local properties of Delaunay triangulations.

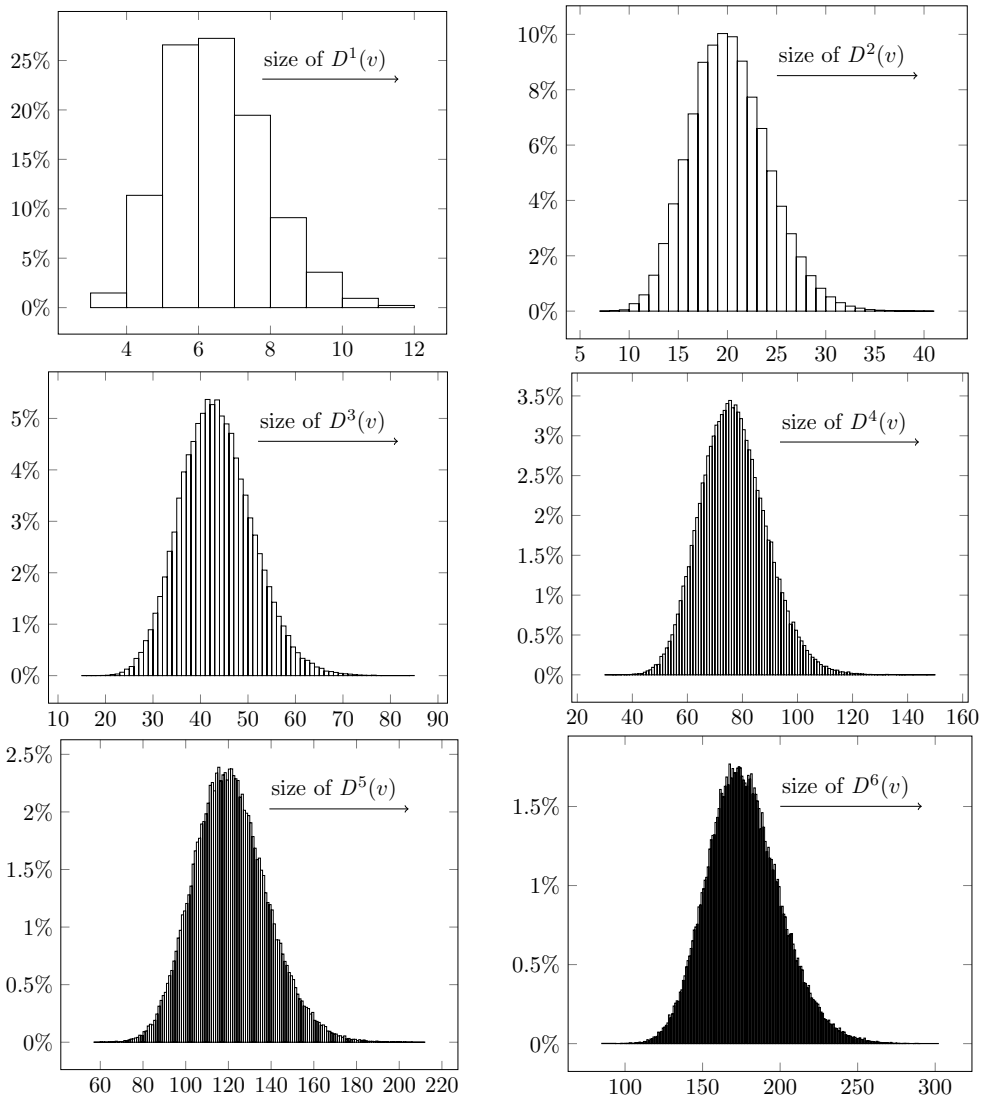


FIGURE 4.3: Distribution of the number of neighbors for D^k

(iv): The definition of a Delaunay triangulation states that no vertex is inside the circumcircle of any triangle. This condition is therefore satisfied by D^1 and consequently by D^k for any k.

(v): The minimum number of necessary neighbors for social interaction varies depending on the intended application and features. This condition is satisfied but, as Figure 4.3 shows, for some social activities it should be safe to take $k > 2$ or $k > 3$. For the beta release of Kiwano we have chosen $k = 3$ which yields a minimum of 15 neighbors and guarantees at least 30 neighbors for 96% of the avatars.

Providing a symmetric relation, the composition with another symmetric relation preserves this property. This is especially convenient when one needs to create the 'fog' of the virtual words, that is, to limit the visibility capacities to a fixed distance. Perceiving the Delaunay neighbors in a fixed range distance is, indeed, a symmetrical relation.

However, another property, we haven't stressed enough yet, is primordial: The chosen neighborhood relation, D^3, should be computable in a distributed manner and scalable when the number of avatars grows. We cover this subject in the next section.

4.2 Data structure

To make Kiwano scalable, it is fundamental to distribute the computation of D^k. To do so, we dynamically maintain a distributed Delaunay triangulation of the avatar positions. In this section we explain how we construct and maintain a distributed D^k graph or DD^k. We then describe the algorithms allowing a dynamic set of avatars, frequently updating their position, to have their neighborhoods computed in a distributed manner among the nodes of the system, a system we call the DD^k overlay.

In this section we present the distributed algorithms and procedures to keep fast neighborhood updates.

4.2.1 Distributed Delaunay^K overlay

Each node i of the DD^k overlay *hosts* a set a of avatars: The node receives the position updates from these avatars and is responsible to update them with their neighbors. Avatars are represented by vertices in the Delaunay triangulation at their respective positions. The set of vertices of the hosted avatars is denoted H_i or, when no confusion is possible, simply H. Each avatar is hosted by only one node so the sets H_i are disjoint two-by-two.

We extend the definition of neighborhood for sets of vertices. For U, a subset of the vertices of the global graph D^k, the neighborhood of U is the union of all neighborhoods of its vertices:

$$D^k(U) = \bigcup_{u \in U} D^k(u)$$

We denote by *core* the set of vertices that have their neighborhood included in H.

$$Core_i^k = \{v \in H_i : D^k(u) \subseteq H_i\}$$

Correspondingly, In_i, the *in-border* set is constituted by the vertices that have neighbors outside of H_i:

$$In_i^k = H_i - Core_i^k$$

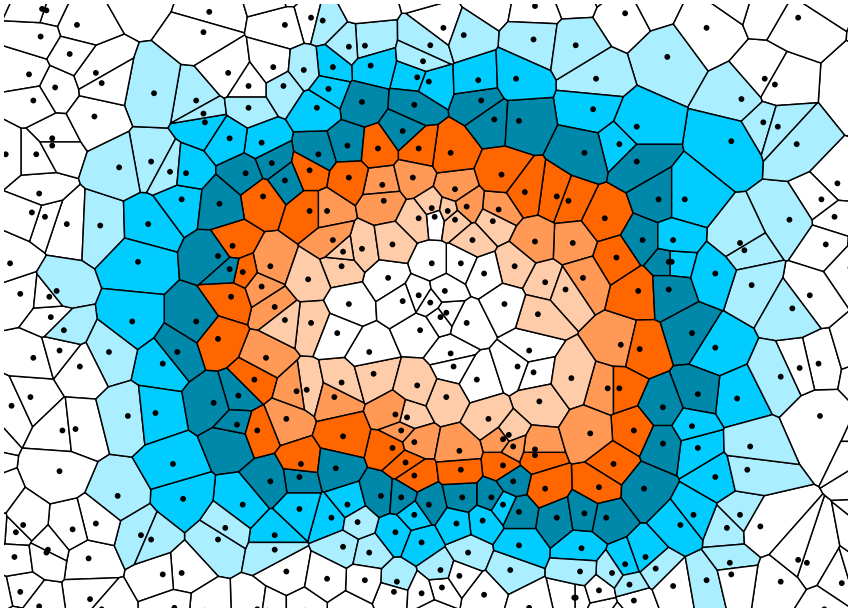


FIGURE 4.4: A zone with the *core* in white, the *in-border* in amber and the *out-border* in blue

To supply to the hosted vertices all their neighbors we need to correctly compute the neighborhood relation for all vertices in H_i. More precisely, we must consider some of the vertices outside of H_i, namely the neighbors of the vertices on the in-border. We name this set *out-border*:

$$Out_i^k = D^k(H_i) - H_i$$

The set of all vertices of node i, that is, the *vertices indexed* by the node, is noted V_i:

$$V_i^k = D^k(H_i) \cup H_i = Core_i^k \uplus In_i^k \uplus Out_i^k$$

As Delaunay triangulations can be represented by their dual Voronoi tessellations, we define the *zone* of a node as the area composed by the Voronoi cells of its hosted vertices. Figure 4.4 pictures a zone and its out-border for $k = 3$. The $Core^3$ is in white, and the three levels of amber correspond to In^3, In^2 and In^1. The out-border is in blue, colored with three different tones for the three different levels. For a Delaunay graph on a sphere, a snapshot of 4,000 avatars hosted by 6 nodes is depicted in Figure 4.8. Each zone has a different color with its in-border in a darker tone.

Each node i computes locally a Delaunay triangulation of its vertices V_i and the corresponding power graph D_i^k. We note $D_i^k|_{H_i}$ the D_i^k relation –and the derived graph– restricted to the range H_i:

$$D_i^k|_{H_i} = \{(u, v) \in D_i^k : u \in H_i\}$$

Every single neighborhood relation i defined by the restricted $D_i^k|_{H_i}$ becomes thus asymmetric. For the ordered pairs (u, v), $u \in In_i$ if and only if $v \in Out_i$. However, for each node, all its in-border objects belong to at least one other node's out-border. In other words, $u \in In_i$ if and only if there is at least one node j such that $u \in Out_j$.

Granted, the following result proves that the computation of the D^k neighborhood relation is distributable.

Theorem 1 (Distributed Delaunay^K graph)**.** *The global Delaunay triangulation graph D^k is the union of the locally computed $D_i^k|_{H_i}$:*

$$D^k = D_1^k|_{H_1} \cup D_2^k|_{H_2} \cup \ldots \cup D_n^k|_{H_n}$$

This is a consequence of the locality of Delaunay triangulations, see Chapter 2, Lemma 1. We can certainly conclude that the neighborhood relation is distributable among the nodes of the overlay.

Also, as sets H_i are disjoint two-by-two, so are the relations $D_i^k|_{H_i}$. However, this is not the case of the sets V_i. Actually, they may overlap on their borders. When $V_i \cap V_j \neq \emptyset$, nodes i and j are *connected* in the overlay and are said to be *neighboring* nodes.

Since the global D^k relation is symmetric, if $V_i \cap D^k(H_j)$ is not empty, then neither $V_j \cap D^k(H_i)$ is empty. In this case we say that D_i^k and D_j^k are neighbors. Correspondingly, the two DD^k nodes i and j are said to be neighbors.

4.2.2 Maintaining a dynamic self-adaptive data structure

The *load* of the entire system depends on all the indexed vertices. For each node i of the overlay, we can define its load:

$$load_i = f(Core_i) + g(In_i) + h(Out_i)$$

Functions f, g, h are generic functions that stand for the load generated by maintaining the vertices in each set: core, in- and out-border, respectively. Core and in-border avatars send the positions themselves and need to receive updates about their neighborhood. Each node has to forward changes on the borders to neighboring nodes. From the aforementioned locality properties of the Delaunay graph we can remark that the three functions have linear average complexity.

In order to assess the resources needed by the system, we need the total load:

$$load = \sum_i load_i$$

Therefore, since D_i^k is computed over V_i, it is worth trying to minimize the number of shared vertices between the sets V_i. This is because the size for In_i and Out_i directly impacts the load by a linear coefficient.

Intuitively, the aim is having most of the neighbors hosted by the same node. In a highly dynamic environment each move tends to disrupt this property. In what follows we show how the distributed data structure is built and maintained optimally distributed among the nodes in Kiwano. This means minimizing the overlap at the borders and the number of neighboring nodes.

Avatar insertion When an avatar joins the world, it needs to connect to a node in order to receive the up-to-date set of neighbors for that vertex. The role of the *dispatcher*, as depicted in Figure 4.5, is to select the entry node in the overlay.

To select the best matching node, the dispatcher would need the complete image of all nodes' zones. And this can be as costly as having an image of the whole set of vertices. For that matter, the dispatcher maintains a simplified map of the DD^k nodes and selects the right entry with high probability. Each node i of this DD^k map is represented by the barycenter of its hosted H_i vertices. In order to maintain the map, the nodes report regularly to the dispatcher the value of their barycenter. The dispatcher selects then the node whose barycenter is the nearest to the avatar's insert position and forwards the insert query to that node. This behavior favorises a compact shape for zones.

The avatar sends its initial position to the dispatcher, which selects the best DD^k node to handle the avatar. The insertion query is then forwarded to the selected node that will add it to H, the set of *hosted* avatars. The node will, from now on, maintain the avatar's neighborhood and receive future position updates.

Avatar transfer on position update In a dynamic environment avatars move, frequently update their position and thus their neighbors change. When most of the neighborhood spans over another node, the vertex is *transferred* to that node. This behavior minimizes the size of the borders.

Load balancing Avatar distribution can vary and, once they move, some nodes may have more load than others. To avoid having most of the charge handled by just a

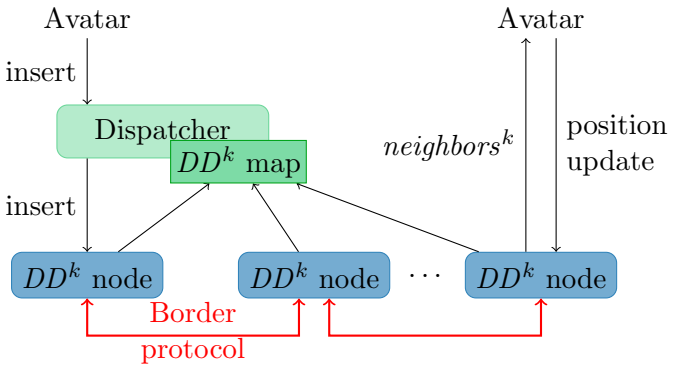


FIGURE 4.5: Distributed Delaunay^k overlay

few ones, nodes regularly send information about their load to their neighboring nodes. When the load of two neighboring nodes significantly differs, the most loaded one transfers to the other node avatar vertices from its in-border until they are balanced. In this way the vertices are step-by-step partitioned into sets of similar sizes. The vertices to be transferred are selected to minimize the distance to the barycenter, again, favoring a compact shape for zones.

Node addition When the total load of the system exceeds the resources available across the overlay, new nodes need to be added. When joining the system, a node must know at least one vertex and its neighborhood in order to be able to maintain the borders with its neighboring nodes. In other words, the smallest zone is the Voronoi cell of a vertex.

A new, empty node queries the dispatcher, which selects the node with the highest load. The selected node partitions the set of hosted vertices H_i in two disjoint subsets $U_1 \uplus U_2 = H_i$ and transfers one of them to the new node. They also communicate the outer borders to maintain the neighborhood in the node overlay. The two subsets U_1 and U_2 are selected using the barycenter criterion to produce two compact sets with little border overlap.

Reshaping Even when the load is similar, the size of the borders may be too large and very few internal vertices. To avoid this case, nodes regularly transfer to their neighboring nodes vertices from their in-border. The vertices to be transferred are selected again to minimize the distance to the barycenter.

4.3 Algorithms

Each DD^k node hosts a set H_i of all vertices. While avatars join, leave and update their positions constantly, the node recomputes D_i^k to deliver, as soon as possible, the neighborhood updates to avatars. This is the *raison d'être* of the DD^k overlay.

We identified two possible manners to compute the D_i^k:

Incrementally. Avatar events –insertions, departures, and position updates– are processed one at a time. Each event independently updates the data structure and all avatars whose neighborhoods are affected are notified about. In this case, avatars receive a message for each modification in their neighborhood.

Periodically. Incoming messages are aggregated and queued for treatment. A new Delaunay graph is recalculated periodically taking into account all collected updates. Finally, all concerned avatars are informed about the changes occurred in their neighborhoods since the last update, that is, the differences between the previous graph and the newly comped one. A similar approach is used in [68] to obtain high frequencies of updates in MOVIES.

We initially considered the incremental method. It seems more suitable for virtual worlds because a short message delivery delay is crucial. For one level of neighborhood, D_1^k, the system performs acceptably. Unfortunately, two or more levels increase the overhead too much. The bottleneck resides in the communication between nodes. There are too many messages to maintain the in and out-borders.

In Kiwano we finally implemented the algorithm for periodical neighborhood computation. If the Delaunay graph is recomputed several, up to ten times per second, the delay in neighborhood updates is hardly noticeable. It is worth mentioning that Kiwano is relevant for discovering who is in the neighborhood. This delay does not apply to updates on known neighbors. Avatar data –including the position– is transmitted to the neighbors by other means, without passing through the DD^k overlay. In the following sections we evaluate the performance with this implementation.

4.3.1 Incremental update

The events treated by a node are generated by the hosted avatars as they enter, move around or leave the virtual world, and by the neighboring nodes for maintaining adequate borders. Each event is treated one at a time and immediately forgotten.

In this case, avatars receive a message for each and every modification in their neighborhood. This method is suitable when avatars do not produce many events and one level of Dealunay neighbors suffices.

Avatar insertion When receiving an avatar insertion from the dispatcher, the entry DD^k node verifies whether the new vertex is inside its coverage zone. If so, the matching DD^k node has been found. The vertex is added to the set H and also to *In* or *Out* if needed. Otherwise, the node forwards the query to another node, the one hosting the nearest neighbor on the out-border. Finally, when reached, sometimes after several hops, the matching node notifies the avatar about its current neighbors and where to send future position updates.

Avatar position update The avatar sends the update to its DD^k node. In addition to updating the internal spatial index with the new position, the node is responsible to notify the neighbors affected by this movement. In case the object was on or is moving to the border, the shape of the zone may change and the bordering servers must be informed of the update. If the vertex crosses the border and goes out of the zone, it is removed from the current node and reinserted as described above.

We make use of the locality properties of the Delaunay triangulations: an object movement affects only the surrounding neighbors of the old and the new positions.

Algorithm 1 Incremental update on each DD^k node

store $oldD_i^k(v) \leftarrow D_i^k(v)$
store old neighbor DD^k nodes for v
for all u in $oldD_i^k(v)$ **do**
  store $oldD_i^k(u) \leftarrow D_i^k(u)$
end for
update v
recompute local changes in D_i^k
for all u in $oldD_i^k(v)$ and $D_i^k(v)$ **do**
  inform u to add $D_i^k(u)$ - $oldD_i^k(u)$
  inform u to discard $oldD_i^k(u)$ - $D_i^k(u)$
end for
update and optimize the border for v

If the modified vertex is on the in- or out-border, then additional operations for border maintenance are necessary, as described in Algorithm 2.

Avatar removal When an avatar leaves, or is discarded after a timeout is reached, the node runs the same algorithm as for position update but considers the set of new neighbors empty.

Algorithm 2 Avatar transfer and border reshape

```
if D_i^k(v) has more neighbors in another neighboring node then
  transfer v to that DD^k node
end if
for all j in old neighbor nodes and not in new neighbor nodes do
  inform j to remove v
end for
for all j in new neighbor nodes do
  inform j to update v
end for
```

Summary At insertion, because we are dealing with a distributed, asynchronous system, the vertex may bounce between two nodes. To avoid this, we impose a node to keep a vertex for a minimum amount of time, to ensure that the borders haven't changed meanwhile. Anyway, as we are dealing with a distributed asynchronous system, we can only reach a weaker form of consistency, attained eventually.

4.3.2 Periodical update

The neighborhood relation can be computed at regular time intervals but, to be sure that changes are notified as soon as possible, in Kiwano we recompute it as soon as the previous computation has finished. Virtual worlds are implemented in a similar way, having a discrete loop in which events that have been accumulated since the last iteration are executed at once [114]. The frequency (or frame-rate [45]) of iterations varies from 10 to 20 times per second. But first, let's see how events (insertion, removal, update) are handled before actually recomputing the D_i^k relation.

Avatar insertion A node creates a new vertex when an avatar is (1) inserted by the dispatcher, (2) transferred from a neighboring node or (3) appears on the out-border. It is then queued to be inserted in the triangulation. When it is inserted or transferred, the avatar is also hosted –its vertex is added to H– and notified about the node that handles future position updates. Otherwise, the vertex is just added to the out-border and used to compute the D^k relation.

Avatar removal When an avatar leaves the world or has not updated its position for a long time, it is queued for deletion.

Position updates When a node receives a position update for one of its indexed vertices, the D_i^k is not immediately recomputed. Instead, the new position coordinates

replace the old ones and will be used for the subsequent D_i^k computation. Position updates for vertices in H come from the avatar itself. For vertices on the out-border the updates arrive from a neighboring node according to the last configuration of the border.

D_i^k computation Before recomputing the D_i^k relation, the set V_i is updated with the pretreated events like border updates or avatar inserts, transfers, and removals. After the computation of the D_i^k relation, all hosted avatars receive notifications about the changes in their neighborhood. Of course, they will receive notifications only if there are any differences, *i.e.*, sets $D_i^k(u)$ - $oldD_i^k(u)$ or $oldD_i^k(u)$ - $D_i^k(u)$ are non empty. Also, neighboring nodes are informed and updated of all vertices that appear to be in their out-borders. The cyclic computation of D_i^k is as follows:

Algorithm 3 Periodical update on each DD^k node

```
loop
  oldD_i^k ← D_i^k
  update the set V_i with the new positions, inserted, and deleted avatars
  recompute D_i^k for V_i
  for all u in H_i do
    inform u to add D_i^k(u) - oldD_i^k(u)
    inform u to discard oldD_i^k(u) - D_i^k(u)
  end for
  do the border update and optimization operations with the neighboring nodes
end loop
```

Summary Vertices of the out-border have their positions updated by the node that hosts them. So, when nodes receive a position update for an in-border vertex, they forward this update to the concerned neighboring nodes allowing them to update the corresponding vertex and the border.

If a node receives a border update for a vertex not already in its out-border, the vertex is added to the set Out. Also, some of the updated vertices may belong to a third node, and this operation maintains the connectivity of the DD^k overlay. We thus ensure coherence at the borders of the DD^k nodes.

The amount of exchanged border messages grows with the size of the in- and out-borders and the number neighboring nodes. This gives another reason to minimize both, the size of the borders and the number of neighboring nodes.

The reshaping process and, in general, the way the avatars are transferred from one node to another are aimed at optimizing these values using heuristics. The preferential transfer of avatars far from the barycenter tends to keep the zone compact and to

minimize its spreading across the space. Then, the computations are not impacted by events occurring in remote locations.

At regular time intervals, the load balancing and node addition processes distribute the load evenly among all nodes. However, every avatar movement tends to break the equilibrium. For some frequency of updates or amplitude of movements the system will witness a degradation in performance. We have measured these performances on our actual implementation and the results of the experiments are described in Section 4.5.

4.4 Architecture

We have seen how Kiwano is capable to handle an unlimited number of avatars in a single, contiguous virtual world and to maintain their neighborhood up-to-date in a distributed fashion. But when the hosting DD^k node changes, the client needs to be aware of the reinsertion. At the application level, clients are not concerned with the distribution of avatars on the nodes.

Kiwano architecture is meant to provide a transparent interface to the client, the infrastructure for a contiguous and interoperable virtual world (for details see Chapter 8).

Let's see how to design the architecture of Kiwano to comply with these requirements.

4.4.1 Architectural transparency with Kiwano

To free the application layer from the distributed internals, Kiwano provides an API[1] to developers [22]. When connecting, a client is assigned a *proxy*, the entry point to Kiwano for the entire session. Each proxy is connected to a subset of all users and, for each of them, maintains the communication with the corresponding node and the neighborhood. This transparency makes Kiwano suitable to be deployed in the cloud.

The DD^k nodes will not be exposed directly through the API. This would lead to too frequent reconnections from the clients. We introduced *proxy* nodes, each avatar having a proxy as single entry point for the entire session, as depicted in Figure 4.6. All proxy nodes report on their availability to the *allocator* which assigns proxies to the newcomers. The proxies forward messages to and from the DD^k nodes hiding the distribution of the Kiwano algorithm from the developers. The connection and interaction with the API is then organised as follows:

[1] Application Programming Interface

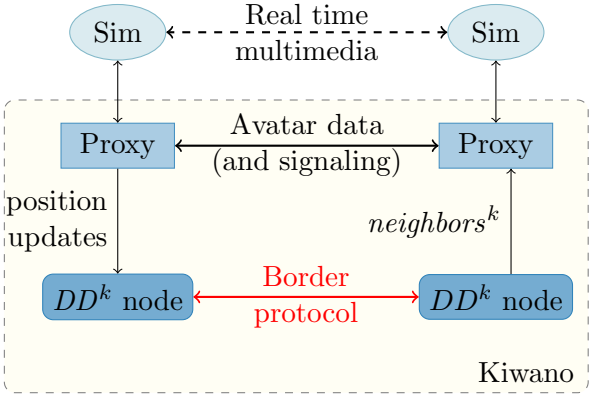


FIGURE 4.6: Kiwano architecture

1: ask the *allocator* for an available proxy (`http GET`)
2: open a websocket connection with the provided proxy
3: [repeat:] send positions updates to the proxy
4: [repeat:] receive neighborhood updates from the proxy
5: close the websocket to leave the virtual world

Proxies are added as and when needed. Their charge depends linearly on the number of avatars hosted. When the number of avatars or the update frequency doubles, Kiwano needs twice as many proxies. The charge fluctuates solely with the insertions and removals. Therefore, proxy scalability is achieved by simply adding gradually enough proxy machines.

The allocator is responsible to keep the load among the proxies balanced in order to have an equal response time for all connected clients. When requested, the allocator chooses to assign the least charged among proxies.

The functioning scheme of a virtual world based on Kiwano is the following: When the user moves its avatar –without interacting with anything– the avatar's position is updated to Kiwano, which returns the updated list of neighbors. In the client's graphical representation the new neighbors are drawn and the former neighbors disappear. If the user intimates an order to its avatars which implies an interaction with its environment, a simulation is completed locally –which produces changes in the scene– and the same order is propagated to the neighbors which compute the same simulation –and should arrive at the same state of the world. However, as simulators do not have the exact same set of avatars and do not receive the exact same messages, they will eventually diverge.

This is where the master replica solution intervenes. In this case, the master replica of each avatar can be hosted in its user's world simulator. And the state of the master replica is broadcasted to its neighbors, forcing eventually the same state in all local simulators, see Figure 4.7.

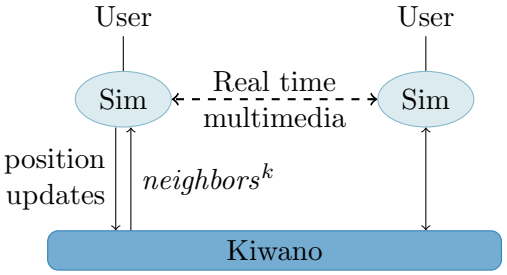


FIGURE 4.7: Distributed simulator with Kiwano

In the last part of the thesis we will present how this architecture has been successfully used to scale a popular MMOG, Manycraft [25, DKV13, VDK13], and to build a mixed reality world, HybridEarth [18, dCDKT14]. Also, we show how it can be used to scale other existing virtual worlds such as Second Life [DK14].

4.4.2 Beta release and public API

After designing the distributed algorithm for computing D^k, we faced the following choice for a first implementation: we could either simulate many nodes in a single thread and slow down the time or implement a nearly final version on many processors. The first alternative avoids the difficulties of distributed debugging and real network communication but adds an error prone layer of complexity with the simulation of distribution.

The second alternative was finally favored since it allowed us real world experiments and gathering of feedback from third party developers. The wide availability of processors in cloud facilities ended up convincing us for a beta release with a public API.

To make sure this first implementation of Kiwano really suits the needs of virtual environments developers, we built it developing in parallel HybridEarth [18, dCDKT14], a virtual world with potential scalability issues. Its design and implementation are discussed in Chapter 8. As this chosen virtual world is a mixed reality world, based on geography, we decided to compute the Delaunay triangulation on a sphere. Recent algorithms [53] make this computation no more costly than its planar counterpart. Also, a sphere is a borderless surface avoiding the degenerated shape of Delaunay triangulations close to the borders of rectangular areas. For virtual worlds built on flat surfaces, the solution is to scale and project that surface on a small enough portion of the sphere. In terms of neighborhood the differences would be hardly noticeable.

Also, to allow interoperability between multi-platform versions of virtual worlds –*i.e.*, running on the three major desktop platforms, the two smartphone ones plus in web browsers (javascript), see Chapter 8– we made the following choices:

- To avoid the difficulty of installing software on the multi-platform end-users' machines, all the nodes of Kiwano are hosted in data centers, *i.e.*, in the cloud.
- As websockets are the only two-way long-lasting connections available on web browsers, the Kiwano API should be accessible through websockets.

The API is open for public experimentation and a more detailed documentation is available on the project's website. We have started collecting feedback from some developers and hope this will improve the usability of Kiwano.

Future plans include the integration of a STUN[2] server in the proxies to help direct realtime communication between simulators.

4.5 Performance evaluation

> *More is different.*
> –P.W. Anderson

To evaluate a contiguous virtual space accommodating orders of magnitude more avatars, we must reconsider the settings to perform the evaluation. Of course, the avatar distribution will differ from the previously presented virtual environments. Now that the perceived space is contiguous their dynamics will also change. But how? In this section we propose a new avatar mobility model and use it to evaluate our implementation of Kiwano accordingly.

4.5.1 Pink Banana avatar mobility model and simulator

Unless it has already reached success, to evaluate the scalability of a system in terms of number of users, one needs to simulate them. To validate its results, this simulation has to reproduce a previously observed behavior. However, in our case, no virtual world ever hosted simultaneously more than a few hundred users in a single contiguous space, and more means different. So, we need to infer by other means what could be the behavior of thousands of users connecting to Kiwano.

Pink banana, our mobility model, takes inspiration from the blue banana model [88] for avatar mobility. The blue banana model infers from Second Life traces that avatars spend most of their time moving slowly and erratically around hotspots. From time to time, avatars travel long distances until they reach another hotspot.

[2]Session Traversal Utilities for NAT

In this regard, the blue banana model does not indicate how the hotspots are distributed in space. We enhanced the pink banana model with geographic information about the distribution of hotspots. We gathered datasets of various human activities [101, 112] that indicate that human population density follows a power law distribution. In our improved model pink banana, we generated a set of hotspots geographically distributed using Lévy flights, thus producing densities following a power law distribution.

Another point omitted by the blue banana model is the distribution in time of the travel events. To overcome this limitation, we used real world data [112]. They emphasize that most human activities occur by bursts, meaning that movement events are distributed in time following a power law distribution. We integrated this in our pink banana model.

From this model we developed a distributed simulator able to generate as many simulated avatars as needed. We first generated a thousand fixed hotspots and all nodes of the simulator were provided with the same set of hotspots, so all the simulated avatars can evolve around the same hotspots. At regular time intervals, each simulator node generates new positions for each of its hosted bots. The interval and the number of bots are parameters adjustable according to the needs of the experiment.

The simulated speed of the avatars is inversely proportional to density of the hotspots. This makes avatars move slower in crowded areas. They tend to stay around their landing hotspot.

Finally, more nodes and hence more avatars can be added at will.

4.5.2 Settings

To perform the evaluation, we disposed of virtual and physical machines of diverse configurations and performances located in four different physical places.

All DD^k nodes, the dispatcher, and the allocator ran on four machines in the same data center. These were 8-core Intel i7 CPU 920 at 2.67GHz, 64bit, with 24Gbytes of RAM.

For running the proxies we had at our disposal

- 15 physical CPUs in a data center, Intel Xeon at 2.00GHz, 64bit, 2Gbytes of RAM,
- 30 virtual machines in a different data center, QEMU Virtual CPU, 64bit, 1.5Gbytes of RAM,
- 6 desktop computers in our office, various CPUs at 32bit, all with 4Gbytes of RAM.

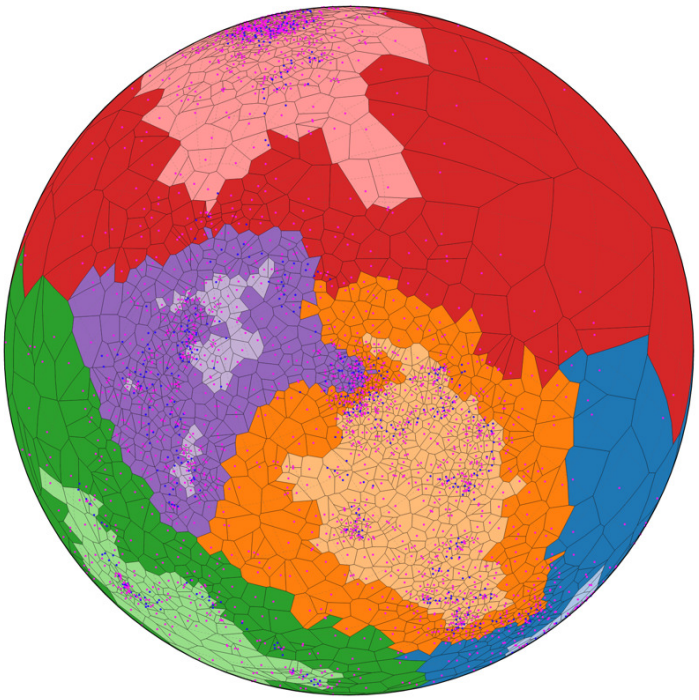


FIGURE 4.8: Graphical monitoring of DD^k nodes

For the sake of simplicity and to automate deployment operations, all systems ran the same flavor of Linux operating system, namely Ubuntu 12.04 LTS.

The avatar simulators ran on modified proxies. These proxies do not accept connections from Sims but otherwise share the same code and behave like the operational ones of the public API, see Section 4.4.1. The proxies reported to the allocator their state, load, and throughput of incoming and outgoing messages. This allowed us to keep an eye on the actual amount of messages to and from the DD^k nodes.

In order to measure their performance and assess their behavior, the DD^k nodes reported to a monitoring server every 10 seconds. The transmitted reports contained:

- the delay between position updates;
- the delay for neighborhood notifications;
- the number of hosted vertices;
- the number of vertices on the out-border;
- the number of messages to and from proxies;
- the number of messages to and from neighboring nodes.

All values are averaged on a 10 seconds period. To verify if the system was running as expected and get insights on what was going on globally, we set up a graphical monitoring, see Figure 4.8. This visual monitor takes at regular time intervals samples of vertices in each DD^k node and shows them graphically on a sphere. Each zone is depicted in a different color. The graphical monitor was deactivated during the measurement sessions to avoid any effect on the recorded values.

In the initial state of the system, we ran the monitoring server, the dispatcher, the allocator and several DD^k nodes. New DD^k nodes can be added to the system whenever they are needed. The dispatcher was configured to add a DD^k node once the average delay between two neighborhood notifications decreased under a fixed threshold.

We launched avatars thousand by thousand and waited several seconds in between for the system to stabilize. During this time, the simulated avatars sent their updates with the programmed frequency. Once there were no more incoming insertion messages from the dispatcher, and the measured parameters became constant, the reports were recorded by the monitoring server for all DD^k nodes at once.

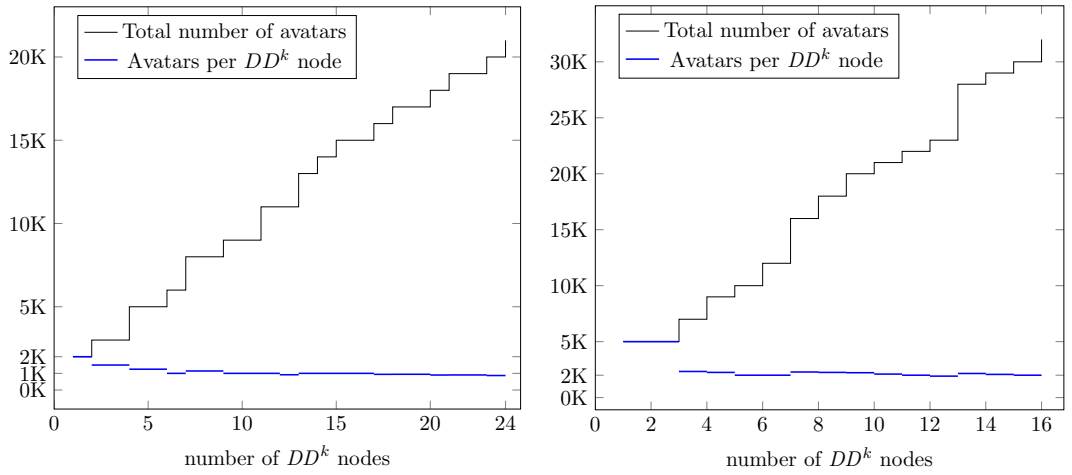
We evaluated two different scenarios:

Action games: Position updates are more frequent, every 0.5 seconds. The delay between two neighborhood updates –when they are necessary– should not exceed 0.20 seconds. If the delay exceeds this value, a DD^k node is added to the overlay.

Social virtual worlds: Each simulated avatar updates their position every 4 seconds. DD^k nodes send neighborhood updates for each hosted avatar. This time, the addition of DD^k nodes is triggered when there are noticed losses of messages to the proxies.

The frequency of updates in the proposed scenarios was evaluated realistic after a previous experiment we ran with a real dataset [98]. This dataset is a capture of several hours of soccer player movements. Although positions were reported every 100ms, most of these movements did not make any difference in the Delaunay graph and, hence, in their neighborhood. Filtering the position updates unlikely to modify the neighborhood will greatly lower the average frequency of actual updates to the nodes with only minor transitory mistakes in the computed neighborhood.

While avatars do not behave exactly like real humans we can expect them to behave so in between teleportations. We didn't implement any filtering mechanisms but we estimate that a wisely designed filter can lower significantly the average frequency of messages needed to be sent to the DD^k nodes. Nonetheless, the proposed frequencies are still useful –even without filtering– for many real world scenarios.



(A) Action game scenario (B) Social virtual world scenario

FIGURE 4.9: Evaluation results

4.5.3 Results

The maximum number of avatars hosted by a DD^k node decreases as the out-border increases. For the first part of the figures, the load, namely, the number of hosted avatars, decreases while the number of neighboring nodes grows. After that, when an avatar teleports, its new position can be over several nodes.

When the number of nodes grows, the probability to go across several nodes increases too. For example, if we had only two DD^k nodes in Figure 4.8, a teleportation, in the worst case, would transfer the vertex to the other node. In the depicted example, the vertex can be transferred subsequently many times before arriving to the matching node. This is why in our measurements the maximum number of avatars per node continues to decrease slightly as the total number of nodes grows. However, since the average degree of the overlay was of the same order of magnitude as the total number of nodes, we cannot assess whether this value stabilizes after some point or continues its slow decreasing.

These first measurements on the performance of a Kiwano implementation give us plenty of confidence in the concept: The reached values in terms of frequency of updates and number of avatars per CPU are already beyond existing systems. This makes Kiwano a valid solution for real world projects. The measured cost per avatar is, as theoretically predicted and expected, nearly constant. We firmly believe that improvement in the algorithms and tweaking in parameters will give even better results.

Also, as the system looked stable in the measured ranges, we can extrapolate the behavior of the current implementation at around a hundred thousand avatars and get an acceptable number of avatars per CPU.

4.6 Summary

Kiwano is a new approach to scale up the number of simultaneous avatars evolving and interacting in a contiguous space. This approach, based on neighborhoods instead of regions, avoids the space partitioning common in virtual worlds and makes possible the implementation of large scale mixed reality universes.

We described the scalable distributed algorithm we designed to compute the neighborhood and presented a complete implementation of this algorithm. This implementation has been released and its public API is being used for virtual worlds software developments [18, 25, DKV13, VDK13, dCDKT14].

Simulations and performance measurements show that this implementation capable to support tens of thousands of avatars with low neighborhood computation costs. These first simulations, where more processors could have been added without collapsing the system, indicates that higher values are easily reachable.

While the software is already usable in real world applications, the design of the upper levels, namely the proxies and world simulators, can be greatly improved. More simulations, by stressing the system, will also help in improving the algorithm and the implementation of the overlay computing the neighborhood.

Also, as Delaunay triangulations are useful in many areas, computational graphics for instance, we can expect our distributed algorithm to inspire further works to take benefit from a distributed computation of the triangulation.

To confirm the validity of the solution and the strength of the implementation, we measured the performances for tens of thousands of simulated avatars running over tens of processors hosted across several data centers. Our evaluation exceeds by orders of magnitude the current state of the art. The results, in terms of number of avatars per CPU, indicate Kiwano to be a cost effective solution for the industry. We expect the scalability brought by Kiwano to open up opportunities for new kinds of virtual worlds. For instance, the dream of a mixed reality world spanning the whole planet with avatars and people –geolocated by GPS– evolving in the same space might soon become a reality [18, 19, 61].

Chapter 5

Kwery: Spatial Containment Queries for Moving Objects

Contents

In this chapter we present Kwery[1], a distributed system designed to efficiently perform spatial queries on large numbers of highly dynamic objects updating their positions up to several times per second.

In Kwery we use a distributed spatial index on top of a distributed self-adaptive tree structure. Each node of the system hosts and answers queries on a group of objects in a zone, which is the minimal axis-aligned rectangle. Objects hosted by a node are chosen based on their spatial proximity and their number varies in order to fit the computational resources of the node. Spatial queries are then answered only by the nodes with meaningful zones, that is, where the node's zone intersects the query zone.

The load of the system is generated by (1) object position update and (2) spatial queries processing. Each node takes care of these operations for its hosted subset of objects supporting the optimal charge while seeking to minimize the covered zone. This is to ensure that each query has a reduced overhead. The resulting load is constantly balanced among the nodes of the overlay.

Under increased load, nodes can be added –or removed– on the fly. This makes our solution ideal to be deployed in the cloud, where the scalability is sustained by taking advantage of the vast computational resources provided.

We evaluated the performance of the system under significant load. We simulated tens of thousands of moving objects, with frequencies of several updates per second. We evaluated the behavior of the critical components, of the load balancing mechanism, and the distribution of the load among the nodes. To estimate the performance on query processing, that is, to see how many nodes will respond in average to a request, we show the resulting overlap between zones.

Kwery is suitable for a large range of applications making use of location-based services. Reverse geocoding consists in finding out what is at a certain location, being a virtual or a mirror world. It is a key feature of Google Maps [35]. In this chapter we focus on the usage in the context of virtual worlds. We employ Kwery as an interest management technique for dynamic objects.

We begin by discussing in Section 5.1 the motivation for spatial queries in the context of virtual worlds. We then present in Section 5.2 the chosen data structure for our spatial index and how to distribute it. We present in Section 5.3 the algorithms to maintain an

[1]Kwery is an ongoing project and the results presented in this chapter have not yet been submitted for publication.

optimal load balance and how to efficiently answer spatial queries. In the last part of this chapter, we discuss how to implement a publish-subscribe system using Kwery. We finally show in Section 5.6.3 how to extend our solutions for more general queries over multiple attributes.

5.1 Spatial queries in virtual environments

Identifying the three game state components [85] –avatars, objects, and terrain– allowed us to separate their management, see Section 1.2. In this way, we are able to apply specific solutions for each scalability problem. We have seen in Section 3.2 that so far, virtual worlds employ for all three components the same fixed zone-based, area-of-interest, approach. However, interest and interaction for objects substantially differ from avatars:

- Objects are not controlled by a human user, nor by a program. They are not 'aware' in the sense that there is not a person that reacts.
- Objects do not 'see' avatars nor other objects. The relations are not symmetric. Objects interact exclusively with avatars.
- Most of them don't move and seem to be just part of the visible décor.
- While avatar to avatar interactions are mostly social, interactions with objects are not. They are used as tools, for a specific and rather limited purpose.
- Objects can greatly differ in size, shape, and distribution.

As a consequence, objects of interest are not necessarily close to an avatar or in the same range-distance. The interest management for objects should allow variations in size and placement of the area. For instance, if Bob is in a desert area he may want to know what's in his wide field of vision, maybe further. However, if he is at the town center, the crowd of objects may obstruct the view, and thus it suffices to obtain information only about the immediate proximity. Also, if Alice, for instance, travels at high speed, she may need to know in advance what is at a certain location.

So, instead of the well established techniques fixed zone-based, area-of-interest, we propose a variable size interest zone. With Kwery, we aim at achieving this by performing spatial range queries.

Of course, this approach makes room for other features in a virtual world. If avatars report their positions, one is able to find out how densely populated is a given zone, or

to monitor the activity at a certain location. And since we are speaking about moving objects, it is important to note that we may also treat avatar positions in the same way.

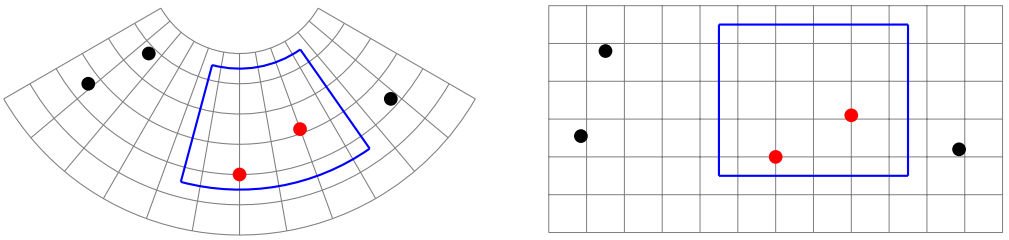
5.1.1 Spatial queries on moving objects

Location information management is a widely encountered problem in computing systems. Retrieving which objects are at a specific location is still a challenging issue, especially when dealing with dynamic objects. We recall here the applications described in Section 2.2.

- *Geographic Information System (GIS)* [30] for querying or monitoring various geographically identified targets. For instance, a tourist information system may allow users to search for attractions in their area.
- *Car fleet management*, for companies to know and optimize at any moment how their resources and demands are spatially and temporally allocated.
- *Local advertising* and *georecommendation* [55]: A typical scenario is sending ads to potential customers in a given area of a local commerce. An example is the restaurant with 20 meals left at 1PM deciding to send 50%-off coupons by text message to phone users nearby.
- *Location (geo) based social gaming* such as Google's Ingress [21]. Other online and social activities are outdoor geocaching [15] or proximity dating, see Tinder [33].
- *Social mixed reality* [18, dCDKT14]: In virtual worlds avatars are mobile objects and spatial queries are issued to determine which avatars and objects are relevant for a given user. In the case of mixed –a.k.a. hybrid– reality, the system must represent uniformly, virtual and real objects.

All these applications need to deal with large numbers of moving objects in a timely manner. In our evaluation, see Section 3.1, we have seen that the performance of current solutions, especially the mostly used spatial databases, is well below the expected demands. Moreover, the state of the art does not cope well with significant variations in parameters, see Chapter 3. In particular, for some applications, meaningful time intervals are minutes –*i.e.*, advertising– while for others, events might occur every few milliseconds –see virtual worlds and social mixed reality. Also, object distribution and mobility patterns may vary considerably in time. With Kwery we aim at allowing all these variations in object behavior and client demands.

Spatial queries involve location and spatial attributes of the represented objects. A typical example is retrieving the objects located inside a polygon. In Figure 5.1 we represent



(A) Polar coordinates (B) Grid coordinates

FIGURE 5.1: Axis aligned spatial queries

graphically a *containment query*. When the rectangle has axis aligned segments, it is also called *range query*. We draw the query range in blue and the contained objects with red. In Figure 5.1(A) objects are represented in the geodesic space, with longitude and latitude coordinates, while in Figure 5.1(B) the same objects are on a plane, with x, y coordinates.

Most virtual worlds are built on a plane grid [28, 32, 37, 60, 91] so, for retrieving the moving objects in the area of interest, spatial containment queries can be reduced to simple range queries. For new virtual and mirror worlds [18, 21, 35] on a sphere, large enough range queries are sufficient.

Unless otherwise stated, in this thesis we employ the simple name spatial query to designate axis-aligned spatial queries, regardless of the coordinate system.

Reverse geocoding or *geolocation* consists of finding out which objects are at a given location. In the case of persistent data the problem is addressed with spatial databases. But objects are numerous and moving, and answering such queries in a timely manner is a challenging issue. We must keep in mind three interconnected properties [51, 64]:

i) *Scalability*: In terms of numbers of objects, frequency of updates and numbers of queries. This is our main concern.

ii) *Consistency*: Queries must be answered according to the most recent state, and the same regardless of the distribution.

iii) *Availability*: Queries must be answered in a timely manner, independently of the number of objects, the frequency of updates, or the distribution model.

5.1.2 Interest management on dynamic objects

Mutable objects, such as doors, tools, and cars, are characterized by their state. Doors can be opened or closed, cars can be racing, frequently changing their position, and tools can be carried, used for interactions with other avatars, like firing a gun towards

an enemy, or for interaction with other objects, like crafting. An object's state can thus be updated –or written– by an avatar, a bot, or a program controlling world's physics, and can be queried –or read– by other avatars.

These objects are often manipulated using databases. In OpenSim, for instance, mutable objects –or items– are associated to a user and they constitute the inventory [29, 91]. User inventories are stored and manipulated using databases through an InventoryService.

In virtual worlds, avatars issue spatial queries to determine which objects are relevant at a certain location. This ensures that a user gets all the information needed to compose the visual field but, to avoid saturation, not much more. Furthermore, the temporal resolution is elevated, for vision is 10 to 30 frames per second, and objects viewed and modified by several avatars at the same time must be responsive enough to provide fast updates to the clients.

With Kwery we propose a flexible trade-off between reads and writes on the spatial index. To do this we use locality in data access. Objects are distributed among the nodes of the system using their geographic proximity.

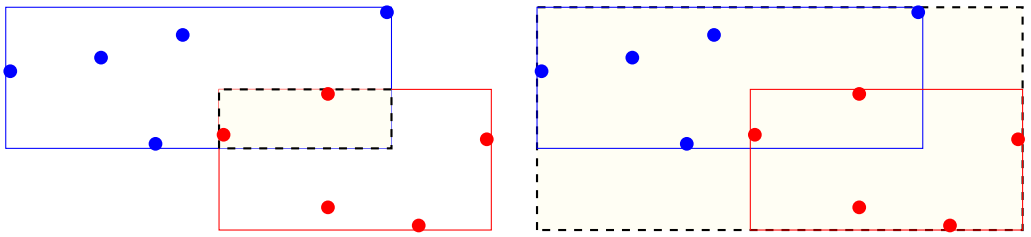
State partitioning: Each object is represented on exactly one node. Hence updates are handled only by the node that hosts the object. This is done by the data structure.

Query execution partitioning: Kwery forwards spatial queries to the nodes that intersect the queried zone. And by minimizing the overlap between nodes, we ensure the query is executed on a reduced as possible number of nodes.

Kwery hides the distribution using specialized nodes for redirecting insertions, updates, and queries. For simplicity, the interest management can be addressed with a convenient index-like interface.

5.2 Distributed data structure

To allow Kwery to scale, the data structure is a spatial index distributed among many nodes. In this section, we explain how our spatial index serves to answer queries about a set of dynamic objects distributed among the nodes of a tree overlay. But first, we begin by presenting how we represent and index the objects in motion.



(A) Intersection of two zones (B) Union of two zones

FIGURE 5.2: Zone operations

5.2.1 Spatial index

To retrieve what is at a certain location, *objects* must be represented in *space*. In our presentation we comply with most virtual worlds [28, 29, 32] and GIS [30], when considering a two-dimensional space. It can be a plane or a sphere. In Kwery we do not make this distinction. We denote by S the two-dimensional –geodesic or Cartesian– space, as depicted in Figure 5.1. For clarity, we will illustrate with planar euclidean space, see Figures 5.3 and 5.2.

In what follows, we specify the core concepts of the spatial index used for Kwery, *object*, *zone*, and *spatial query*.

Objects have location, they are attributed a position in space, their (x_p, y_p) coordinates. We denote by O the set of all objects.

Zones are two dimensional axis-aligned rectangles. Given the coordinates of its bottom-left corner x_z, y_z, and top-right corner X_z, Y_z, a zone z can be easily defined by the predicate encompassing all interior points:

$$z(p) \;=\; x_z \leq x_p \leq X_z \;\wedge y_z \;\leq y_p \leq Y_z$$

A zone can also be understood as a subset of the space, namely all points that satisfy the corresponding predicate. With this approach we can define the union and the intersection of two zones, also depicted in Figure 5.2.

The area covered by the *intersection* of two zones is always a rectangle. In Kwery we also call it *overlapping* and simply denote it by

$$z = z_1 \sqcap z_2$$

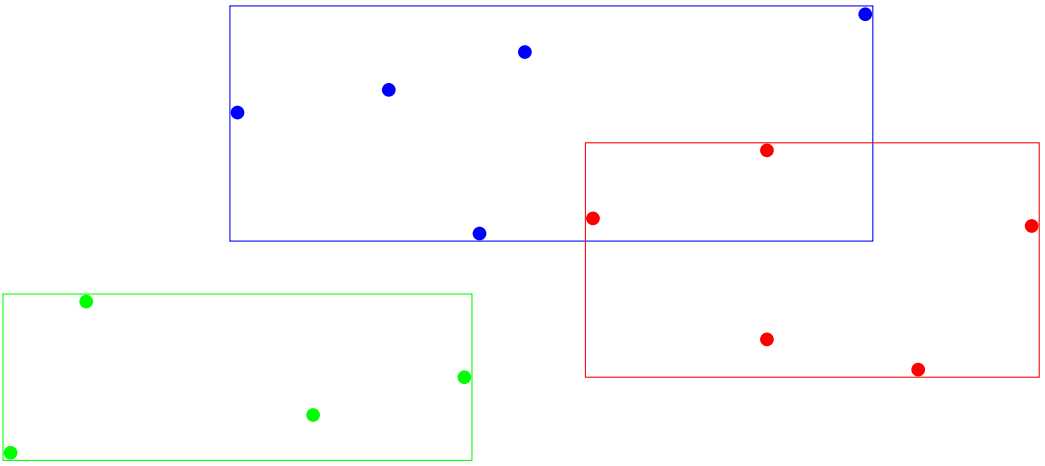


FIGURE 5.3: Three zones covering 14 objects. The red and blue zones overlap.

for any two zones z_1 and z_2. The resulting coordinates are straightforward:

$$x_z = max(x_{z_1}, x_{z_2}), \quad y_z = max(y_{z_1}, y_{z_2}),$$

$$X_z = min(X_{z_1}, X_{z_2}), \quad Y_z = min(Y_{z_1}, Y_{z_2}).$$

When $X_z < x_z$ or $Y_z < y_z$, we say the zone is empty.

On the other hand, the *union* of two zones may not be a rectangle. We denote denote it

$$z = z_1 \sqcup z_2$$

To preserve the aforementioned zone properties, in Kwery we define the union of z_1 and z_2 as the smallest rectangle covering both zones, with the following coordinates:

$$x_z = min(x_{z_1}, x_{z_2}), \quad y_z = min(y_{z_1}, y_{z_2}),$$

$$X_z = max(X_{z_1}, X_{z_2}), \quad Y_z = max(Y_{z_1}, Y_{z_2}).$$

We say an object is *covered* by a zone z if its position is inside the rectangle of the zone or, formally, if its representation is satisfied by the predicate of z. We then denote by $covered_z$ the set of all objects covered by a zone z:

$$covered_z = \{o \in O : z(o)\}$$

Figure 5.3 shows tree zones. The green zone does not overlap with any other zone. The red and the blue do overlap and the two top red objects are comprised.

Spatial queries Since a spatial *query* consists in retrieving the objects in a given zone z, it is therefore formulated as $covered_q$, where q denotes the queried zone. For instance, in Figure 5.1, the same query zone is depicted for polar, or longitude-latitude geographical coordinates (A), and for grid coordinates (B). The covered objects inside are colored in red.

5.2.2 Distributed data structure

The Kwery data structure must allow scalability for two different types of operations:

- *Query answering:* Objects are accessed by querying the system. The answer must be provided timely, before the information becomes obsolete.
- *Object position update:* Highly dynamic objects have their position updated at arbitrary high frequencies. The system must be able to keep the data structure up-to-date, in order to timely answer the queries.

To make Kwery scale, it is fundamental to distribute the two operations. Yet, in Kwery objects have a unique representation and both operations employ the same data structure. The trade-off between these requirements is managed locally, by each node. This is how we ensure consistency eventually.

In this section we explain how we construct and maintain a distributed spatial index. We then describe the distributed algorithms for answering spatial queries. They cope with the objects frequently updating their position and distributed among the nodes of the system. In Kwery, these are called *zone nodes*.

For k zone nodes we partition the set of objects into k disjoint subsets.

$$O = O_1 \uplus O_2 \uplus \ldots \uplus O_k$$

Each zone node i will be assigned one subset of objects O_i. It receives updates for these objects and answers queries that contain them. Each node i will have a corresponding zone z_i that is the *minimal bounding rectangle* that covers its set of objects.

We denote by Z the set of all such zones:

$$Z = \{z_i : covered_{z_i} = O_i\}$$

It is important to note that all zones in Z cover the entire set of objects regardless of the covered area. In other words, they do not necessarily cover the entire space. However,

this is not a key requirement for reverse geocoding because a query for an empty zone that does not have any objects will have an empty result.

An object may be covered by several zones. However, note that the object will be assigned to a unique zone among these, for example the two top red objects in Figure 5.3.

In the following, we extend the data structure to capture the distributed nature of Kwery. We further prove that to obtain a correct result it suffices to evaluate containment queries only on the concerned nodes, that is, on the nodes that have a covering zone intersecting the query zone.

In what follows, we naturally assume that all objects are spatially located and that each one of them is assigned to some zone. Let $z_1, \ldots, z_k$ be a set of zones that cover all indexed objects. For any $obj \in O$, there exists $i \in \{1, \ldots, k\}$ such that $obj \in O_i$, or, equivalently, $\cup_{i=1..k} O_i = O$.

Theorem 2 (Distributed spatial queries)**.** *To know all objects in a given query zone z, we need to process it on the zones that intersect it:*

$$covered(z) = \bigcup_{z_i \sqcap z \neq \phi} (O_i \cap covered(z))$$

Proof. Since we are interested in the indexed objects, we know that $covered(z)$ is a subset of O. We can therefore write

$$covered(z) = O \cap covered(z)$$

From the hypothesis we can rewrite O and we further obtain

$$covered(z) = (\cup_{i=1..k} O_i)) \cap covered(z)$$

Applying the distributivity property of the set operation,

$$covered(z) = \cup_{i=1..k} (O_i \cap covered(z))$$

But, when two zones do not overlap, $z_i \sqcap z_j = \phi$, we know that no object can belong to both of them, that is, $\nexists x, x \subseteq z_i \sqcap z_j$. Then, according to the definition of the zone overlap, $\nexists x$ such that $x \subseteq z_i$ and $x \subseteq z_j$, and thus, $\nexists obj \in O$ such that $obj \in O_i$ and $obj \in O_j$. We can conclude that indeed, the evaluation can be performed only on the zones that have a non-empty overlap with z, therefore

$$covered(z) = \bigcup_{z_i \sqcap z \neq \phi} (O_i \cap covered(z)) \qquad \square$$

This is an important result providing the formal ground for the space division and global index distribution into multiple local indices. Theorem 2 shows that it suffices to evaluate containment queries only on the zones that have a non-empty intersection with the requested zone.

The position update of an object that has previously been absent corresponds to an insertion. Also, when the updated representation is outside the current zone, the system must decide to which zone the object is mapped. The new spatial representation could be covered by several zones, by one zone, or by no zone at all. We should be able to measure which is the best zone to assign an object. For this we define a *geometrical score*

$$score(o, z)$$

for an object representation o and a zone z. Depending on the application requirements, the assignment of an object to a zones may be done according different criteria.

Systems that deal with frequent position updates need an even object load. On the other hand, systems that need efficient query answering do not afford high overlap rates. For instance, a system that deals with random object movements and does not deal with numerous queries may be willing to trade any overlap requirement for a good object load balancing.

5.3 Algorithms

Kwery provides an index interface, so external requests are issued by (1) the moving objects when updating the index, and (2) by the clients by querying areas. These two kinds of sources independently charge the system. Kwery dynamically redistributes the load among the nodes to match the dynamic trade-off between these sources of load.

We now proceed to the presentation of the algorithms that answer external requests. Then, we explain the self-adaptive procedures used in Kwery to redistribute the load in a hierarchical infrastructure.

5.3.1 External requests

As a spatial index, Kwery provides update and query primitives. The moving objects update the index with insertion, destruction, and position update in the virtual environment. Additionally, clients issue spatial queries. Here we spell out the algorithms that take care of these external events.

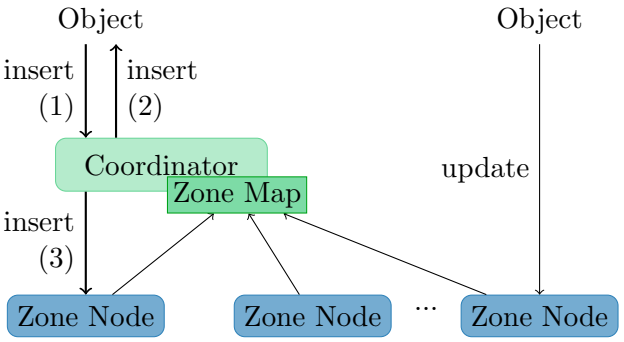


FIGURE 5.4: Insertion of a new moving object

Object insertion When an object is created it is assigned a position in the virtual world. To be visible, it must be inserted at a zone node. The role of the *coordinator*, as depicted in Figure 5.4, is to select the best matching zone node for an incoming object. Object distribution among zone nodes can dramatically impact the overhead for query answering.

To minimize the zone overlap, the coordinator maintains a *zone map* of the nodes in the system. It consists of the zones covered by each node, *i.e.*, the predicates for each zone. The map is regularly updated by the nodes when it shrinks, and by the coordinator itself, when the zone enlarges at insertion.

Using the zone map, the coordinator selects the node whose zone maximizes the overlap according to the *geometrical score*. Choosing at each step the best object distribution is too costly, and therefore, we chose the nodes locally, and compute the score based on geometric heuristics. If the position is covered by several nodes, the coordinator selects the zone with the least charge. But if all covering nodes are saturated, a close node will expand its zone to cover the new object's position. Finally, the coordinator informs both, the object and the node.

As an entry point, the coordinator is theoretically the unique bottleneck of Kwery. However, it only deals with assignments. We assess the practical limits of one coordinator in Section 5.5 and propose a hierarchical architecture scalable vertically in Section 5.6.1.

Object destruction On its corresponding zone node, an object is removed from the set of covered objects when it has been explicitly destroyed –*i.e.*, with a specific message– or when the update timeout is reached –*i.e.*, position updates have been omitted for an extended period of time.

Object position update Objects send their new positions to the corresponding zone node. Each node incrementally updates its local index. A position update can change

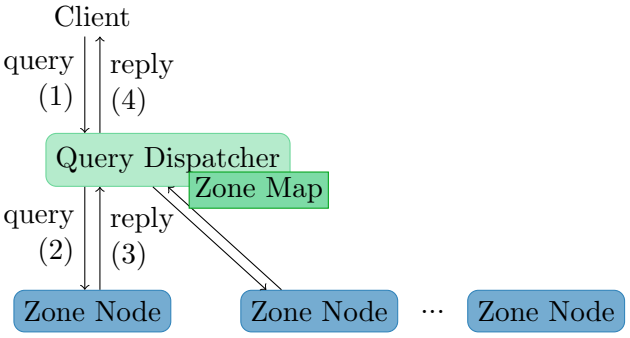


FIGURE 5.5: Spatial queries distribution

the node's zone by enlarging or by shrinking it. If the new position is already covered by the zone, the local index is immediately updated. Otherwise, the object is sent for reinsertion to the coordinator. When an update changes the covered area, either by shrinking or by enlarging it, the node is responsible to update all copies of the zone map.

Spatial containment query A *client* addresses containment queries to a *query dispatcher*. Just as the coordinator, the query dispatcher maintains the zone map. The architecture is shown in Figure 5.5. Using the map, it forwards the queries only to the nodes whose zones intersect the query zone. Nodes evaluate the same query on their local indexes and reply back to the dispatcher. The dispatcher aggregates these answers before responding back to the client.

Since each object is held by one node only, replies can simply be forwarded back to the client with no supplementary processing. However, as we are speaking about a distributed system, queries are not guaranteed to be evaluated at the exact same time at every node. Therefore transitory glitches may appear, for instance an object to be be returned by two nodes. And here intervenes the aggregation process of the dispatcher.

The complete procedure ran by the dispatcher is described in Algorithm 4.

One query dispatcher may be seen as a limitation for a number of applications dealing with many queries. Still, the query dispatcher can be replicated, each replica maintaining the same map and taking care of some queries. Our proposal for a vertically scalable architecture is presented in Section 5.6.1 and 5.4.1.

5.3.2 Internal dynamic self-organization

In virtual words, the geographical distribution of objects and the frequency of their updates change in time. Also, the size and the distribution of spatial queries varies.

Algorithm 4 Query dispatcher: Spatial containment query

```
loop
  for all (q, u) query q from client u do
    compute the zones that intersect the query zone zones_q ← {z : z ⊓ q ≠ φ}
    for all z in zones_q do
      forward query (q, u) to z
    end for
  end for
  for all query answer covered_z(q, u) do
    aggregate covered_z(q, u) to covered(q, u)
  end for
  send covered(q, u) to client u
end loop
```

Each zone node is responsible to maintain the positions of its objects up-to-date and also to timely answer all incoming spatial queries. Therefore, a good load balancing technique must allow for a flexible trade-off between these two tasks on each zone node.

We explain in what follows the elements allowing a flexible system sizing. Our hierarchical structure employs a zone map to reduce the overlap, and therefore to reduce the overhead created by query processing. The role of the coordinator is to keep the overlap as small as possible while ensuring at the same time that no node becomes saturated. As we will see, the load balancing provisions the space with zone nodes according to the local densities.

Zone map update To summarize, the zone map is held on the coordinator and on all query dispatchers. Each node is responsible to update all copies of the map as soon as its zone coverage changes.

One may chose for instance to update the zone map only sporadically or at regular time intervals, to reduce the bandwidth and CPU consumption. However, queries are not guaranteed to receive all objects when zones became larger meanwhile. Otherwise, if zones shrink significantly, they may receive queries that do not concern them and unnecessarily increase latency and overhead.

This kind of optimisation is, however, dependent on the intended application. They are therefore subject to other future discussions and investigations.

Zone node addition A scalable, self-adaptive solution in a heterogeneous system must be able to add and remove nodes at any time. For an on-the-fly node addition, several techniques can be applied. One of them could be similar to that described for the Kiwano algorithm in Section 4.3, where a new node receives half of the load of the

most loaded zone node in the system. A partitioning algorithm can ensure the summed area of the two resulting zones to be smaller than the area of the initial zone.

In Kwery we decided to create a new node with an empty zone. As soon as the coordinator has to (re)assign an object the new node will be selected. The load balancing mechanism will eventually assign more objects and distribute the total load.

Load balancing The system's self-organization is performed on the fly, at each update, with respect to the geometrical score. Each zone node informs the coordinator about its total load, that is, the load generated by the queries and by the index update. A node will not enlarge its zone unless required by the coordinator. As for the coordinator, it will require a node to enlarge its zone only if it is not completely saturated. In this way, each node will handle exactly its capacity, without being oversaturated.

5.4 Architecture

We have seen how the positions of mobile objects are distributed and dynamically transferred between the nodes of the system in order to maintain a reduced overlap and fast query responsiveness. Ideally, the system is transparent to users and provides the image of a contiguous space. We describe in what follows the architecture to provide a uniform interface to clients similar to that of an index.

Kwery is a general solution to answer range queries, and universal extensions are presented in Section 5.6.3. Specifically, in this thesis we focus on scalability for dynamic objects in virtual worlds and in the second part of this section we explain how Kwery can be employed to this purpose.

5.4.1 A transparent, contiguous virtual space

To provide a unified interface to the user, we describe here a similar architectural approach to the one presented in Section 4.4 for Kiwano.

In Kwery the zone nodes are not accessed directly by the objects and the clients. We introduce *proxy nodes*, each of them maintaining the connection for a group of objects and/or clients, as depicted in Figure 5.6. They provide a transparent interface similar to that of a spatial index with insertion, removal, update, and range query primitives.

Proxies forward incoming insertion messages to the coordinator, update and deletion messages to the assigned zone node, and spatial queries to the query dispatcher. As they

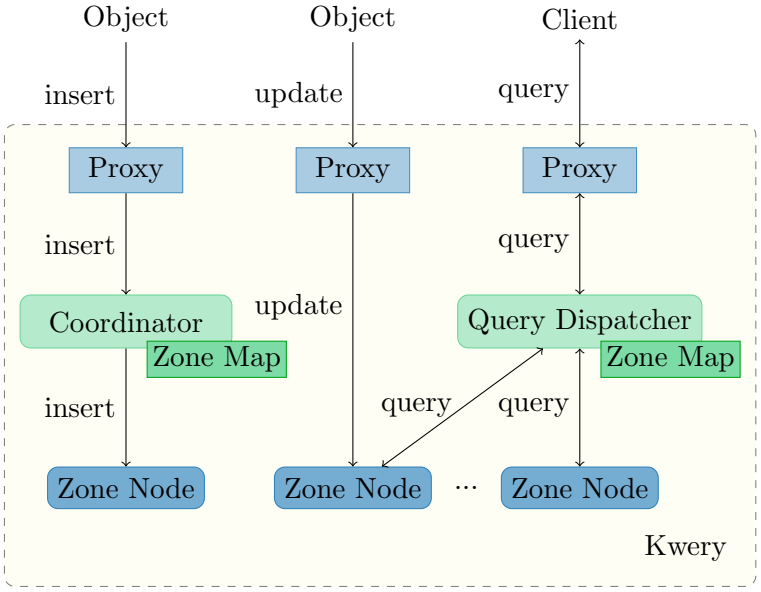


FIGURE 5.6: Kwery architecture

do not retain information about the objects and the queries, their charge is generated solely by the incoming messages. Thus, their scalability is linearly dependent on the traffic from the users and clients. They can be added as and when needed.

Given that queries are independent of each other, the query dispatcher can be replicated as much as needed. From an architectural point of view, this improvement only needs a mechanism to replicate the zone map at each dispatcher and disseminate map updates.

Behind the scene, Kwery is an index distributed among several nodes. Nodes do not communicate directly and the transfer of objects is made via the coordinator. Zones are basically independent indexing structures, supplied by the proxies and the coordinator.

This independence permits zone nodes to vary their local indexing structure. Furthermore, this makes Kwery a scalable, distributed architecture for any incremental spatial index.

5.4.2 Object management with Kwery

In virtual worlds the system's load is generated by the mutable elements, namely, avatars and objects. We have seen that most of it comes from position updates, 70% as presented in [57]. In centralized implementations the unique simulator takes in charge the whole load generated by avatars and moving objects, but they are very limited. In a distributed system, the load is taken care of by multiple machines. However, to provide the impression of a contiguous space, many of them either use replication [49] or force the client to connect to multiple instances at the same time [91].

It is worth noting that, unlike Kiwano, there is no redundancy for the indexed objects. Each one has a unique representation on a zone node at a time. However, if we chose to scale vertically by adding multiple layers of query dispatcher and coordinator nodes, we obtain a tree structure. Thus, there we will have several layers of coordinators and dispatchers that take care of object nodes.

Also, unlike Kiwano, objects and clients are different entities. This clearly illustrates the asymmetry in interaction between avatars and objects and answers the different requirements for interest management.

The self-adaptive nature of Kwery allows variations in object dynamics. Each event concerning an object, namely, position update or query, generates extra charge. Nodes use this exact measure to dynamically redistribute the objects when needed.

5.5 Evaluation

The distributable data structure and the algorithms for Kwery are promising concepts. Thus far, we have implemented a prototype supporting all aforementioned algorithms. This first implementation served us mainly to validate the proposed architecture and also to test and assess the utility of the heuristical geometrical score. These make room for future perspective presented in Section 5.6.

Granted that queries –in regard with size, frequency and distribution– are very much conditioned by the purpose of the application, it is of little interest to effectively evaluate them. What we can do instead is to measure the overlap rate as an indicator of the resulting overhead. Indeed, as previously discussed, the performance of the query processing depends on the number of nodes concerned by queries. And the overlap rate between the covering zones directly impacts system performance.

In this section we present the evaluation of our implementation by simulating significant load, tens of thousands of moving objects updating their positions several times per second. We evaluated the behavior of the critical components, of the load balancing mechanism, and the resulting overlap between nodes as a practical measure of the performance of the query processing.

5.5.1 Setup

We implemented Kwery in Python. We used Twisted framework for network communication, json encoding to serialize messages, and pyrtree –a python implementation of R-trees– for objects on each node's local index, and for the zone map on the coordinator.

Zone nodes have been deployed on an 8-core Intel i7 CPU 920 at 2.67GHz, 64bit, with 24Gbytes of RAM located in a data center. Each zone node process runs independently on one CPU core.

The load was generated on the proxies, each simulating some moving objects.

In each simulation, the coordinator and the query dispatcher started first. They waited for incoming connections from proxy and zone nodes. When all proxies and zones have started, objects were launched by the proxies and began moving.

At regular time intervals, the coordinator aggregated and logged the information from all nodes. Each node constantly reported to the coordinator the following:

- *The zone* covered by the node, as the bottom-left and top-right coordinates. All values aggregated constitute the zone map.
- *The barycenter* of the points assigned, as described below.
- *The load* as the percentage of the CPU utilized by the process. This value corresponds to the total load handled by the node.

As the initial churn rate has a significant impact on the accuracy of the evaluation, all our measurements were performed in stable states, with no moving objects joining or leaving the system. To do so, we waited several seconds for all objects to be assigned to a zone, before actually logging the measures. It's worth noting that the load generated by practical, routine, insertions and deletions is not particularly meaningful as they are handled like the reassignments from the points of view of the coordinator and the nodes.

5.5.2 Parameters

In our tests we assumed a planar euclidean space with objects represented by their x, y coordinates.

First, let's discuss the coordinator's assignment policy. It is, so far, the unique entry point for object insertion. Therefore, to effectively treat a large number of node assignments, it needs to be light and efficiently implemented. We remind the reader that the coordinator computes the geometrical score between an object and a zone node. As discussed in Section 5.3, a minimal score results into a fair trade-off between overlapping and position updates.

When receiving an assignment request, the coordinator uses the distance between the position of the object and the center of a zone z, in order to incrementally minimize the overlap. We investigated two policies for keeping the barycenter c_z:

- The center of the zone rectangle (as geometrical shape) computed by the coordinator

$$x_{c_z} = \frac{1}{2}(x_z + X_z), \qquad y_{c_z} = \frac{1}{2}(y_z + Y_z)$$

- The barycenter of the points assigned to the zone, its coordinates are incrementally calculated by each node

$$x_{c_z} = \frac{1}{N}\sum_{p \in O} x_p, \qquad y_{c_z} = \frac{1}{N}\sum_{p \in O} y_p$$

  Where O is the set of objects assigned to z and N is the cardinal of O.

In our implementation the coordinator selects the zone node whose barycenter is closest. If there are several, an arbitrary one will be chosen among those that are not completely saturated. We recall here that an empty zone is always preferred for insertion and reassignments. However, if all close zone nodes are saturated, then the coordinator chooses an arbitrary node.

Indeed, in this setting the coordinator is theoretically the unique bottleneck of Kwery. However, at the moment, the limits of the system are dictated by the movement pattern of the objects. Let us explain. If object teleportations occur frequently, as of every movement, nodes need to request reinsertion at each position update. This means that the coordinator manages the complete load resulted from updates. However, this case is unlikely to appear in real or virtual world scenarios: it disrupts the world coherency.

The coordinator therefore handles just a fraction of the total position updates. In this respect, we propose in Section 5.6.1 a scalable hierarchical architecture. The idea is to allow multiple coordinators to be managed by a *supercoordinator*. A coordinator will address the supercoordinator to relocate a zone node when its charge is too big.

The object distribution determines the zone coverage and the map. There is a considerable amount of work that studies human and avatar distribution and dynamics, see [88, 89, 108, 110]. Yet, on general object distribution and events studies are limited [94].

Recent works [109, 114] find that in Second Life the number of objects per region is roughly constant. They also note that object distribution resembles avatar distribution with 30% of the regions have less than 100, and 65% between 100 and 1,000. However, objects tend to move less and to maintain a roughly constant distribution.

We therefore chose to evaluate two different scenarios for object initial distribution.

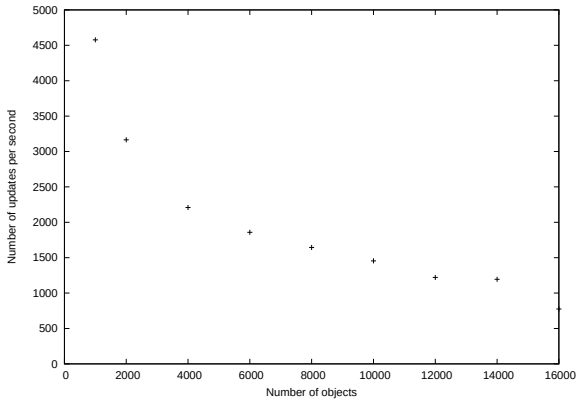


FIGURE 5.7: Maximum number of updates per second handled by a single node versus number of objects

- *Uniform*, corresponding to a uniformly occupied world by independent moving objects. When they do not communicate, and do not form hotspots, this is an extreme but expected configuration for objects.
- *Power law*, corresponding to a common human and avatar distribution. The moving objects may be either the avatars themselves or other objects of their interest.

We are now ready to move these objects. For many application scenarios the characterization of object dynamic is still a challenging research issue [94]. For that matter we employed a simple model of movement, inspired by previous works on avatar dynamics. This model corresponds in particular to the movement of avatars in virtual worlds [88]. In our simulations each object independently follows a Lévy flight.

In the following we varied the number of objects, the number of zone nodes, and the distribution of objects' positions.

5.5.3 Single node's load

Kwery builds a distributed index using several local indexes. Hence, its global performance depends on the performance of each local index. To size the system deployment and determine its practical limitations, we first ran simulations on a single zone node.

Figure 5.7 shows the maximum number of updates our system can handle per second with a single node. The overall shape of the curve corresponds to the complexity of the R-tree structure used to index moving objects. It shows the logarithmic complexity for insert, remove and retrieve operations, as we evaluated them in Section 3.1.

With 2,000 objects, a single node can handle 3,100 updates per second. Each object can thus update its position 1.55 times per second, that is every 0.65s. With 12,000 objects,

each one can update its position 0.1 times per second, that is every 10s. Moreover, given a fixed update frequency, Figure 5.7 indicates the maximum number of objects a zone node can handle. For example, if objects update their positions every second, a single node can handle up to 2,700 objects.

5.5.4 Coordinator's load

The practical load of the coordinator is threefold. Firstly, it updates the zone map. Secondly, it receives assign from zone nodes trying to reduce their loads or proxies receiving new objects. Thirdly, it sends messages when objects are actually reassigned to a different node to balance the load.

Figure 5.8 shows how coordinator's load evolves with different numbers of nodes for the two investigated object distributions. Roughly, each node is assigned a thousand objects. On the X-axis we represent the number of nodes –note the exponential progression– and on the Y-axis, the average number of messages received by the coordinator from one node per second. Each bar is composed of three segments:

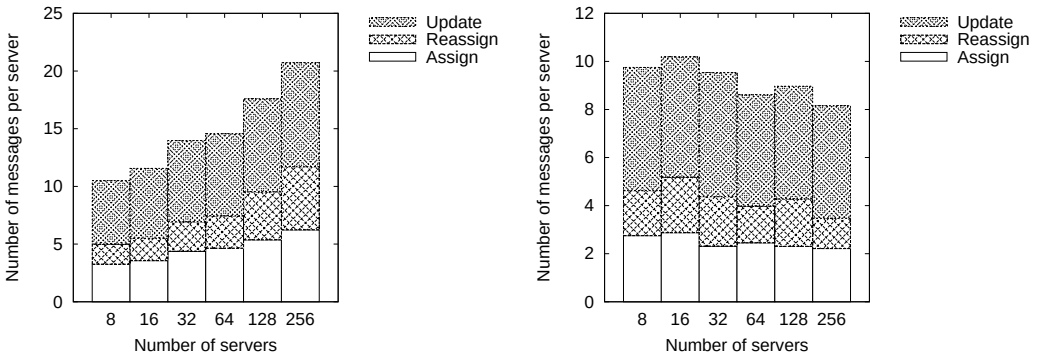
- *Update* is the number of zone update messages the coordinator received,
- *Assign* is the number of reassignments solicitations the coordinator received,
- *Reassign* is the number of actual reassignments performed by the coordinator.

Of course, the number of assign queries is greater or equal to the number of performed reassignments ($Assign \geq Reassign$).

When moving objects are uniformly distributed, see Figure 5.8(A), with 256,000 moving objects distributed amongst 256 nodes, the coordinator receives in average 21 messages per second from each node.

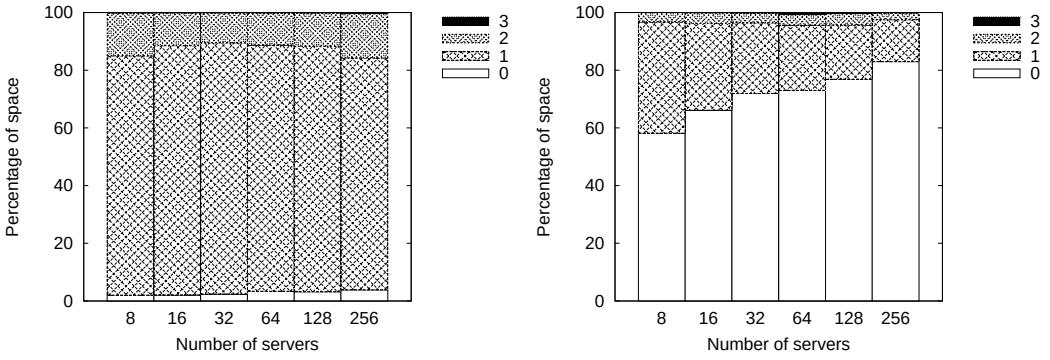
When moving objects are uniformly distributed, see Figure 5.8(B), 256,000 moving objects distributed amongst 256 nodes, the coordinator receives in average 8 messages per second from each one.

Processing any *Update*, *Assign*, or *Reassign* event is logarithmic in the number of nodes because the coordinator employs the same R-tree structure to maintain the map. In a system with 256 nodes, the index of the coordinator stores 256 entries. Given the previous result for one node in Figure 5.7 we can safely assume that the coordinator can handle at least 5,000 events per second. As a consequence, in a system of 256 zone nodes handling 256,000 moving objects, each of them can update its position every:



(A) Uniform (B) Power law

FIGURE 5.8: Coordinator's load



(A) Uniform (B) Power law

FIGURE 5.9: Overlap rate

- 1s if their initial positions distribution is uniform;
- 0.4s if their initial positions distribution is a power law.

5.5.5 Overlap rate and query performance

Kwery maintains a dynamic coverage of the populated space. Each node is responsible of a dynamically evolving zone. One of the core roles of the coordinator is to avoid this overhead by minimizing the overlap between zones such that queries to be processed on as few nodes as possible.

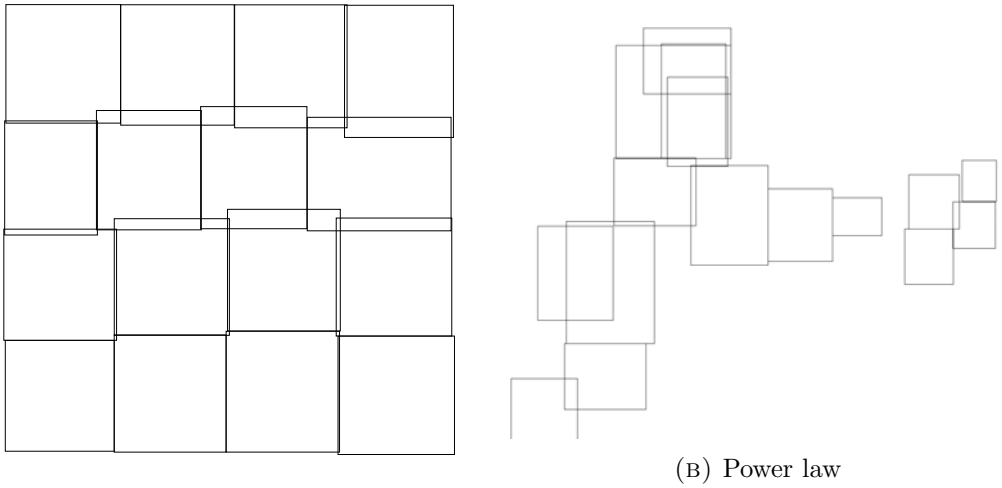
When objects positions follow a power law distribution [112] 80% of the space is empty. When the number of nodes increases, the granularity of space coverage by their zones increases as well, approaching that limit.

Figure 5.9 shows the measures on the overlap rate for several numbers of nodes. On the X-axis we represent the number of nodes and on the Y-axis we represent the percentage

of occupied space. We use the same settings we used to test the load for the coordinator. In this setting too, a node hosts a thousand objects. Each bar composed of (up to) four parts. The first part indicates the percentage of space covered by no zone, the second part the percentage of space covered by exactly one zone, and so on.

With uniform distribution of moving objects initial positions 5.9(A) most of the space is covered by one zone. Intuitively, given the Brownian nature of movements, the average density of moving objects remains constant. Moreover, the uniform distribution of positions and the coordinator's assignment policy tend to make square zones. Figure 5.10(A) is a snapshot of the covering obtained after three minutes with 16 nodes and 16,000 moving objects.

Figure 5.10(B) is a snapshot of the covering obtained after three minutes with 16 servers and 16,000 moving objects.



(A) Uniform

(B) Power law

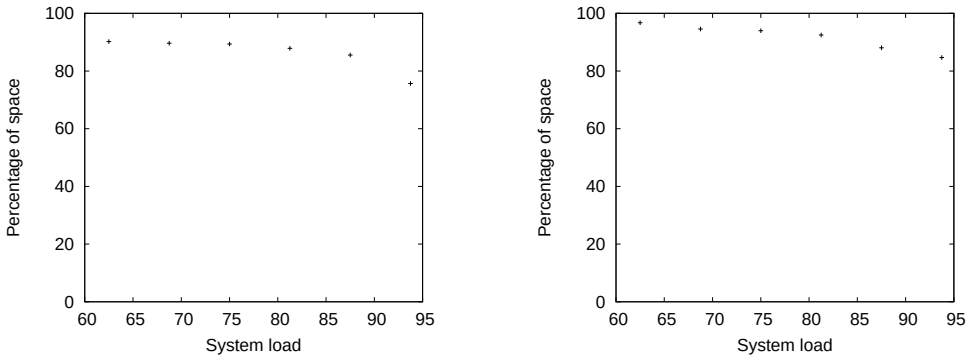
FIGURE 5.10: Space coverage with 16 servers and 16000 moving objects

5.5.6 Overlap under increasing load

Coordinator's strategy to assign moving objects to nodes is a trade-off between two concerns: geometrical ones (minimizing the overlap) and load ones (balancing load). When the number of moving objects increases, the coordinator is more likely to make choices that are not geometrically optimal.

In the following we consider a system with 32 machines, each one being able to handle 1,000 moving objects. The system is can be 100% loaded with 32,000 moving objects.

Figure 5.11 shows the impact of load on overlap rate. X-axis is the system load in percentage, Y-axis is the percentage of space covered by at most one zone. With both



(A) Uniform (B) Power law

FIGURE 5.11: Overlap rate versus system load

uniform –Figure 5.11(A)– and power law –Figure 5.11(B)– distributions the percentage of space covered by at most one zone remains superior to 80% when the system load reaches 90%.

5.6 Extensions

Kwery is a very promising architecture and the preliminary evaluations presented above support this claim. These unmatched performances encourage us to envisage future perspectives and to work towards them. In this section we discuss how we plan to extend the ideas presented so far to provide complete solutions.

Firstly, we address the vertical scalability of the hierarchical architecture. In this way, we overcome the possible bottleneck imposed by the coordinator. Secondly, we describe how this system can be utilized to offer a simple publish-subscribe interface for virtual worlds. And finally, we generalize the mathematical model to find larger limits for a distributable data structure to perform multi-attribute queries.

5.6.1 A hierarchical architecture

In our proposed architecture the coordinator is the unique component to assign zones to objects. As we have seen, its practical limits are comfortably large, but they still depend on the number of reassignments. Also, the size of the zone map will, at some point, become inconvenient for the coordinator and the query dispatchers as well. Indeed, for only one level of subordination, the complexity for the zone map matching grows with its size.

Our proposed architecture is adapted to support unlimited vertical scalability. A coordinator has its own coverage zone as the union of the zones composing the zone map.

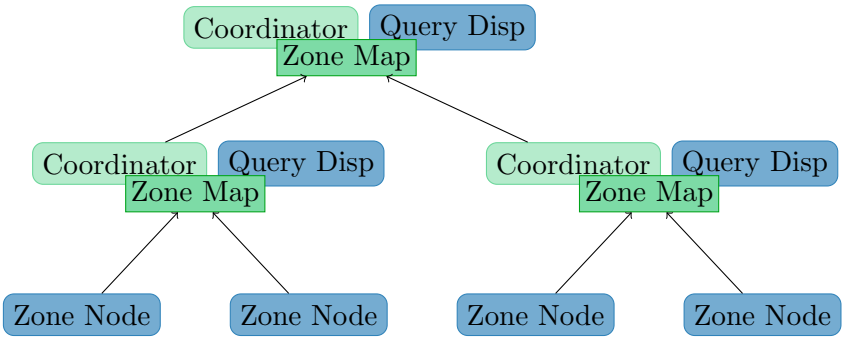


FIGURE 5.12: A hierarchical Kwery architecture

We consider objects' representation to be on level 0 and zone nodes on level 1. Several such independent coordinators on level i can be 'coordinated' by a node on level $i+1$ in the same fashion zones are. A depiction is showed in Figure 5.12. Coordinator nodes on level i report to their coordinator on level $i+1$ their coverage zone and their load.

An object insertion visits exactly one node at each level. Reassignments reach up in the tree until the position is matched by at least one node.

To answer spatial queries we proceed as for the classical distributed R-tree. Each node has its own query dispatcher(s) holding the same zone map. A query is sent to a descendant query dispatcher only if its zone intersects the query zone.

5.6.2 Implementing a publish-subscribe system with Kwery

To retrieve objects at a certain location, Kwery employs spatial containment queries. So far, we know that, to be constantly updated with the changes, one can issue the spatial query with the desired frequency. It is clear that this procedure generates useless overhead. And this is the reason virtual environments prefer to implement publish-subscribe systems. The idea is to allow clients to subscribe to a zone and receive updates each time an event occurs.

Publish-subscribe systems often rely on spatial indexes or spatial databases [44]. We remind you that range queries and publish-subscribe are largely interreducible. Updates simply correspond to publications. Subscriptions are more elaborate than queries, queries are discarded as soon as they are answered while subscriptions imply that all future modifications will be notified until the subscription is cancelled.

With Kwery we resolve subscriptions with a distributed and self-adaptive index. Each zone node can store the incoming queries and check for new updates whether they match any subscription. In Kwery, zones change their shapes very frequently. In fact, each position update of the objects on the border triggers an update for the zone map.

Therefore, subscriptions need to be reevaluated each time one of the concerned zones changes. When the zone shrinks the node may simply discard the unwanted subscription zones. When the zone is enlarged the query dispatcher must intervene and inform the node about the new subscriptions that match.

This work is very promising because it constitutes the first spatial publish-subscribe architecture suitable to be deployed in the cloud with massive scalability.

5.6.3 Minimal requirements for a distributed data structure

Position updates are the main source of computation in virtual environments [57]. The complexity of the problem arises from the need to perform multi-dimensional attribute queries on dynamic objects. In fact, this problem is not specific to two dimensional geographical environments.

In this section we propose a general model to represent objects in a multidimensional space. We explain here how to distribute it onto many nodes in a way to perform multi-attribute queries efficiently. Let's reconsider the main notions of the distributed data structure.

Objects As we just mentioned, objects are represented in an *n-dimensional space*, let it be S. In what follows, we employ the term location to designate an instance in the space S.

Objects are therefore present at a location, they can occupy a polygon, a point or they can be unconfined. Yet, for practical concerns, it is required that their intersection and inclusion to be computable. We denote the set of objects by $O \subseteq 2^S$.

Zones A zone is a predicate over all points in the space.

$$z : 2^S \rightarrow \{true, false\}$$

A zone must be understood as a characteristic function for the points in S. It should also be understood as the part of the space it covers, namely the set of points that satisfy the predicate z.

$$z = \{p \in S : z(p)\} \subseteq S$$

We will prefer the latter notion and consider it intuitive if we are speaking about ranges and contiguous portions of the space.

Since we need to compute intersection and inclusion between zones and objects, zone inclusion must be computable too. The largest possible zone contains all points in space $true(S)$ while the smallest will be the empty zone *false*. We note the set of all zones with Z.

The *union* of two zones $z_1 \sqcup z_2$ can be any zone thus defined in a way that comprises the two, that is:

$$z_1, z_2 \subseteq z_1 \sqcup z_2$$

Dually, the intersection or the *overlapping* of two zones $z = z_1 \sqcap z_2$ is true only for the points where z_1 and z_2 are true; otherwise stated, the zone z comprises all points in z_1 and z_2.

We can define the coverage of a zone in the same way as before. An object is *covered* by a zone z if its representation lies completely inside zone or, formally, if all points inside its representation are satisfied by the predicate of z.

$$covered_z = \{o \in O : z(o)\}.$$

The important result expressed in Theorem 2 holds with the extended data structure as well, that is

$$covered(z) = \bigcup_{z_i \sqcap z \neq \phi} (O_i \cap covered(z))$$

This result shows that it suffices to evaluate the queries only on the zones that have a non-empty intersection with the requested zone. We can therefore conclude that the object assignment and query processing in a multi-dimensional space are distributable among the nodes of the overlay.

We deem of interest as a future perspective to investigate formal properties of a multi-dimensional distributed data structure. In applicative scenarios as those described in Section 5.1.1 for example, spatial attributes may be combined with other preferential ones. For example a matching score for dating [33] or difficulty to find a geocach [15].

5.7 Summary

In this chapter we introduced Kwery, a distributed system to perform spatial queries on large numbers of highly dynamic objects updating their positions up to several times per second. Kwery already shows us that scalability for timely reverse geocoding in virtual worlds is feasible.

Kwery answers –to various degrees– to the three requirements we stated in the first part of the chapter: data scalability, consistency, and availability, as presented in Section 5.1.1. Condition (i) is ensured by the large number of zones that can be simultaneously coordinated but also by the fundamental vertical scalability of hierarchical architectures. Condition (ii) is achieved by having a single representation of the object. Because we maintain data distributed, transitory inconsistencies may appear. However, the system stabilizes and provides some eventual data consistency. And finally, we remind that each node handles the trade-off between object position updates and query answering locally, granting the high data availability of Condition (iii).

Our proposed architecture is scalable both in terms of objects numbers and frequency of updates. We have shown via extensive simulations that a cluster of 256 servers stands up to 256,000 objects moving every second.

While we employed in our implementation R-trees to for each node's local index, Kwery makes no assumption on the nature of the local indexes. Zone nodes may vary their local indexing structure. And this makes Kwery a scalable, distributed architecture for any incremental spatial index.

To our knowledge, Kwery is the first massively self-adaptive solution for reverse geocoding in virtual environments suited to be deployed in the cloud. We emphasize once more that the massive scalability became feasible once we separated the dynamic components of the world. Having mobile objects decoupled from the geographical space allowed us to treat them separately, and thus, provide specialized solutions.

Part III

Architecture and Applications

Chapter 6

MMOG Case Study: Manycraft

Contents

> If a tree falls in a forest and no one is around to hear it, does it make a sound?

Minecraft [28] is a popular game with more than 20 million paying users and many more playing the free version. However, in multiplayer mode, only a few thousand users can

play together. Our measurements show that, even reducing the landscape, *i.e.*, the map, to a uniform flat land, a server cannot host significantly more users.

For a common use case, when players cannot modify the map, we have designed and implemented Manycraft [25, DKV13, VDK13], a distributed architecture to scale the number of users together in the same Minecraft map. Minecraft protocol messages are of three kinds: control, entity and map. In this approach, Kiwano, our distributed infrastructure for scaling virtual worlds, takes care of the entity related messages while the others are processed by a Minecraft server assigned to the player.

In our architecture, the Minecraft server and the message translator are placed on each player's machine. Together they constitute a node in Manycraft to which the client connects as to a normal game server. All nodes share the preloaded map and access the neighborhood through Kiwano.

We begin this chapter by presenting the context in which Minecraft is positioned, as a relatively recent game but enjoying huge success. We present in Section 6.2 a description of the game with an emphasis on its multiplayer mode. Since the source code of the game is not public, we designed and ran performance tests and the precise settings and results are related in Section 6.3. Our main contribution is the design of Manycraft, a novel cloud-based solution to allow an unlimited number of users to play together and interact in a Minecraft map, and is presented in Section 6.4. The implementation developed in our team and available at `http://manycraft.net` is described in Section 6.5. Finally, we conclude the chapter with the ongoing work to enhance and extend our solution to cover more events for a rich user interaction with the environment.

6.1 Minecraft as research challenge

Minecraft is a sandbox construction game [28], that is, users build their own world. Players do not have specific goals to accomplish; They choose how to play the game. Minecraft has forgone realism for creativity and simplicity. This is probably the recipe of its huge popularity: Estimations attribute Minecraft more than 20 million paying users and many more using the free editions.

Minecraft is a game about breaking and placing blocks. At first, people built structures to protect against nocturnal monsters, but as the game grew, players worked together to create wonderful, imaginative things [28]. One of the main interests of Minecraft is the creative mode where players build their own world driven by the pleasure of construction.

(A) WesterosCraft [92] (B) Notre Dame de Paris

FIGURE 6.1: Notable Minecraft productions in creative mode

WesterosCraft [92] recreates the fantasy world of Game of Thrones, see Figure 6.1(A). Notre Dame de Paris in Figure 6.1(B), the Empire State Building, the Enterprise Spaceship and countless real and imaginary places have been produced. Some can be visited but most are only visible in online videos. This is a first challenge: To allow anyone to visit seamlessly a Minecratft map.

While it is mostly a single user game, Minecraft has a multiplayer mode where users connect to a server in order to interact and communicate with each other within the same world. However, servers are strongly limited in the maximum number of simultaneously connected users: the most powerful ones reach only a few thousands [14].

The source code is not available, and the user scalability problem has been attributed –rather dogmatically– to bandwidth and throughput limitations. Trying to better understand what impedes scalability, we conducted experiments with thousands of simulated users, enough to saturate any server resource. Our simulations show that bandwidth consumption, CPU load, and memory usage significantly increase with the complexity of the world and the number of connected players.

Unexpectedly, with a minimalistic map, a uniform flat land for instance, and totally inactive players, all resource utilization remains high and the maximum number of players supported by a server does not increase significantly. Therefore, we can deduce that merely maintaining the presence of the players is a resource intensive activity. Our evaluation discussed in Section 6.3 shows clearly that the map does not generate a considerable charge in Minecraft.

The second self-imposed challenge is to allow un unlimited number of players to be and interact in the same Minecraft map. And since Kiwano is able to do this without relying on the world to scale, we show that with Manycraft, maps can be available for anyone to roam inside freely.

When we began investigating Minecraft, we found no academic paper in the computer science area. Not only the game was relatively new –the first version launched in 2009– but also the source code is closed.

Having initially a unique developer, its technical design is pragmatic and classical: it is simple, with few and specialized protocol messages, coarse pixelated graphics and trivial game physics. This very same simplicity has attracted a lot of tinkering activity: users have reverse-engineered the protocol, decompiled the client and the server. Finally, modifying the software has become part of Minecraft gameplay [97], and users give superpowers to their characters or invent new rules. Particularly, Bukkit [11] software is an add-on to the Minecraft multiplayer server to *mod* –the short for modify– its standard behavior. This allows, for instance, protocol analysis and technical enhancements.

This is what draw our attention and eventually the academic interest of new research community [70]. And this is what makes possible to investigate and propose solutions for scalability in multiplayer mode. All in all, we judge a new on-line game, with simple graphics and a closed source code an ideal research challenge.

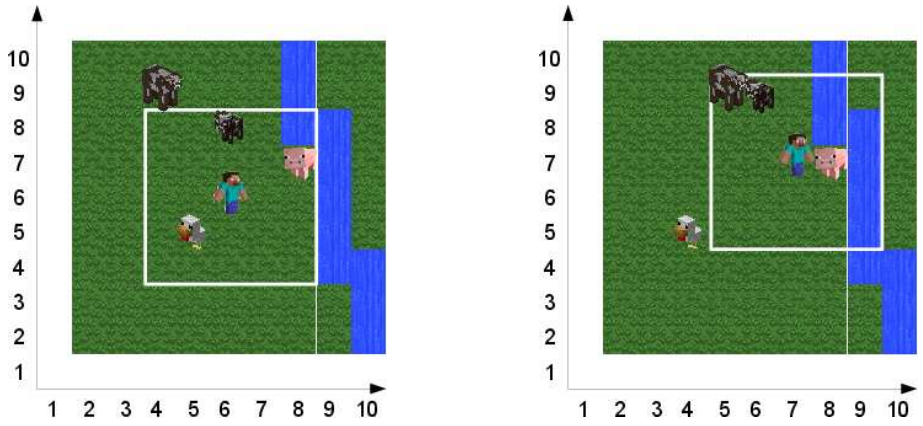


FIGURE 6.2: Between two consecutive ticks Steve is notified about the changes in his area of interest: the chicken left, the calf moved north, and the cow appeared.

6.2 Minecraft architectural aspects

A Minecraft *map* is made of simple cubic *blocks*. All blocks have the same size and vary only in their type (wood, coal ore, stone, etc.). Players are represented as two block high avatars usually known as *entities*, like Steve in Figure 6.2. Have a look at our team's group photo in a Minecraft world in Figure 6.10. Around its position, each entity has a square-shaped area of interest providing the visible map and the perceived *neighbors*.

Non-player characters are called *mobs*[1] and they are affected by the environment in the same way as players, but no data about the map or the neighborhood is transmitted.

In multiplayer mode all players run a Minecraft client and to play together they connect to the same server. Let's examine the current client-server architecture of Minecraft.

6.2.1 The server

A Minecraft server hosts the world and delivers the content to the connected players. The server's time unit is the tick, which lasts 50ms. Within each tick, the server updates world physics, mobs' behavior, and notifies each player about changes that occurred in its awareness area. A brief example is depicted in Figure 6.2. The elements constituting the map evolve. Dynamics such as water flowing, plants growing, or night and day alternation, are computed when they are part of someone's field of vision. Also, mobs are simulated and they react to the environment by moving, attacking, and so on.

6.2.2 The client

When the connection to the server has been established, the player is spawned at a position chosen by the server. During the game, the client sends to the server player actions, namely position updates, block destruction, etc., and he is notified back about events occurring in his area of interest.

6.2.3 The protocol

With each new version, Minecraft evolves and so does its protocol [26]. The first release of the game at the end of 2011 had 19 message types and this number increased up to 83 by the end of 2013. Meanwhile, the game added new types of entities, new skins, different control for small and large movements, prediction, and so on. Albeit the significant evolution, the protocol messages can be divided into the following categories: map, entity, and control.

- *Map.* Map related messages –list of blocks with descriptions– are generated when a player arrives at a new position or when blocks in the area of interest are modified. The server sends only the differences with respect to the last configuration.

[1] short for 'mobile'

- *Entities.* When a player or a mob moves, the new position is send by the server to every player that should be aware of it. Entity related messages also bear chat, avatar animation, or other entity actions.
- *Control.* Periodically, the server sends keep-alive messages to clients. There are also cyclic events, such as day-night alternation, that generate messages at regular time intervals.

In Bukkit, each protocol message sent or received by the server generates one or more events. To be able to intercept them and modify the server's behavior upon their arrival, the most convenient way is to add specific handlers. More details that helped us for scalability, such as message format or how to intercept them, are presented in Section 6.5.

6.2.4 Modes and their scalability requirements

The game can be played in either creative, adventure, or Player Versus Player (PVP) mode. The PVP mode is centered on player interaction often with bellicose gaming experiences. In adventure mode, the objective of the game is defined by world's creator, and players mostly interact with other players and dynamic objects in the décor.

In creative mode, the purpose is to build things by adding and removing blocks. As it can be played offline, in single-player, this mode is very popular. But currently, in order to present their creations, players either host a server on their own computer and allow others to connect, or record a video and share it on the Internet. The first solution is cumbersome and costly: The user needs to maintain a server for a few friends that might visit, while the second one does not offer to those interested the possibility to actually visit the world and be there.

Interestingly, Minecraft provides different mechanisms to limit or even forbid players to modify blocks. Moreover, many popular gaming scenarios tend to emphasize player-to-player interactions. For instance in role playing or racing games, the map is not meant to be modifiable, it constitutes the static décor.

So for all these scenarios where the map is static, player-to-player interaction prevails. We propose Manycraft to allow scalability in the number of users in the same map. Furthermore, with Manycraft, we allow users to make their maps available for anyone to roam inside freely.

6.3 Evaluating Minecraft scalability

Looking at online Minecraft multiplayer servers [14], we can see that the maximum number of simultaneous players is in the few thousands. However, it is not clear what impedes to go further. Our hypothesis is that the mere presence of players is enough to explain this limit, and having a map reduced to the minimum does not push the limit by orders of magnitude.

The source code of the Minecraft server is closed and the communication protocol reduced to the minimum, hardly readable. But we know some architectural elements: The server hosts the world and computes the physics; It also decides which actions are allowed, and computes what is in each one's area of interest. What is more important is that a multiplayer session is hosted on a single machine. Therefore, as users connect, the server eventually runs out of resources. But what are the limits and what do they depend on?

Since the source code is not available, we had to stress the system as a black box. We reduce the map to a uniform flat land: a layer of rocks where on top of which the entities move freely. Also, the players are myopic, the area of interest is reduced to the minimum.

In this setting we evaluate the throughput, the memory usage, and the CPU load as the number of avatars increases. Let's see which one of them is the bottleneck!

6.3.1 Experimental setup

To estimate the load generated by the players' presence we reduce both environment complexity and in game interactions. Firstly, the world contains no mobs, only blocks and players. Hence, the server does not have to compute any behavior for the mobs. Secondly, the map is *Superflat* [27] and made of a single layer of rocky blocks. As a consequence, we basically eliminate the physics of the world, as no plants are growing, no water is flowing, and so on. Thirdly, our simulated players don't chat, nor mine, nor craft. Their only actions are position updates. Fourthly, we set the view distance to its minimum value in order to reduce the traffic as much as possible. This is a well-known hint to reduce the server's load [20].

We ran our experiments on four identical machines hosted in the same data center. Each machine had an 8-core Intel Xeon at 2.8GHz, 24 Gbytes of RAM, ran Ubuntu 12.04, Oracle Java 7, and were connected together through gigabit Ethernet. The average latency between machines is around 0.8ms.

One of the machines ran the CraftBukkit [11] server –a Minecraft server modded to accept Bukkit plugins– to be evaluated. The other machines ran several simulated players using text-only Minecraft clients, with no graphical rendering.

Players' movements followed the model of avatar movement in virtual worlds described in the blue banana model [88]: a player is either slowly randomly walking or moving fast in a given direction. Our simulated players move at the same speed as regular –*i.e.*, human driven– Minecraft players and, similarly, send position updates 20 times per second.

6.3.2 Measurements

In this setting we measure the bandwidth throughput, the memory usage, and the CPU load. As we are interested in finding out what impedes the sever to scale, our measures are always according the number of users.

In our experiment we add players to the server one by one, waiting 5 seconds between two insertions. This delay avoids players to have their connection denied by the server due to an instant heap of connections. After slowly inserting a batch of one hundred, we wait several minutes for the system to stabilize before taking the measure. This ensures that the initial cost of entering the world does not impact the measurements for resource utilization.

Bandwidth throughput

Figure 6.3 represents the attained server throughput in megabytes per second for an increasing number of simulated players. These two curves clearly show that the bandwidth consumption grows linearly with the number of players.

Each Minecraft client sends ∼3.5Kbytes/s and receives ∼3Kbytes/s. For a server connected to the backbone through gigabit Ethernet –a common setting for servers in data centers– we can assume a full duplex with maximum throughput of 500 Mbits/s. This indicates that bandwidth should start to be a limiting factor for approximatively 20K players.

With a more complex, modifiable and interactive map the throughput per user should be greater and the maximum number of player significantly reduced.

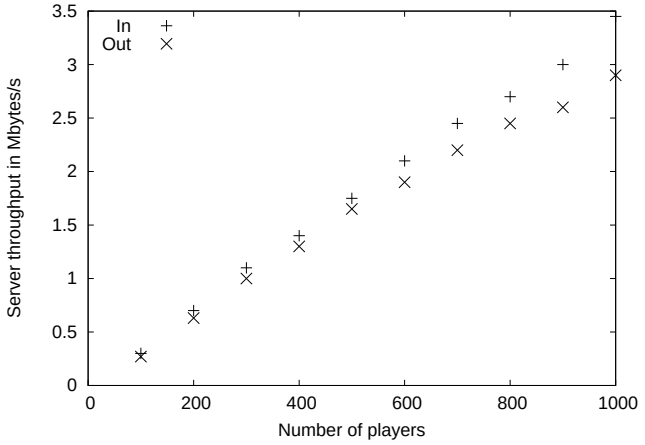


FIGURE 6.3: Throughput

Memory usage

In Figure 6.4 we show the amount of RAM used by the server in Gbytes. Given that the server runs on a Java virtual machine, the slight deviations may result from the behavior of the garbage collector.

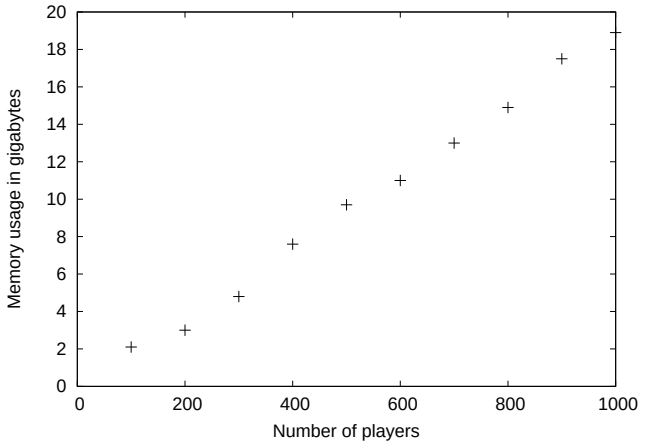


FIGURE 6.4: Memory usage

At this scale, the memory usage of the server can be approximated to grow linearly with the number of connected Minecraft clients. A machine capable to allocate 24 Gbytes to the Minecraft server reaches, therefore, its memory limitation for approximatively 1,500 connected players.

CPU load

To evaluate the impact of entity dynamics we performed the simulations using moving entities but also still players, *i.e.*, they connect but remain inactive. Figure 6.5 shows

the results of the two measurements: The number of players is represented on the x-axis, while on the y-axis we have the aggregate percentage of the CPU load for all cores. A percentage above 100 indicates that more than one core is used by the Minecraft server. Given that we used an 8-core server, the maximum load it can stand is 800%.

We were not sure that our model for avatar movement would reflect the actual behavior of Minecraft players. But the comparison of the CPU load, all the avatars moving vs. motionless avatars, shows, unexpectedly, that the mobility patterns have little influence on the CPU load of the server.

Both measurements show that the CPU load can be approximated to grow linearly with the number of players in the range supported by one server. Therefore, with a very simple map, the maximum load supported by our 8-core CPU should be just above 1,000 players.

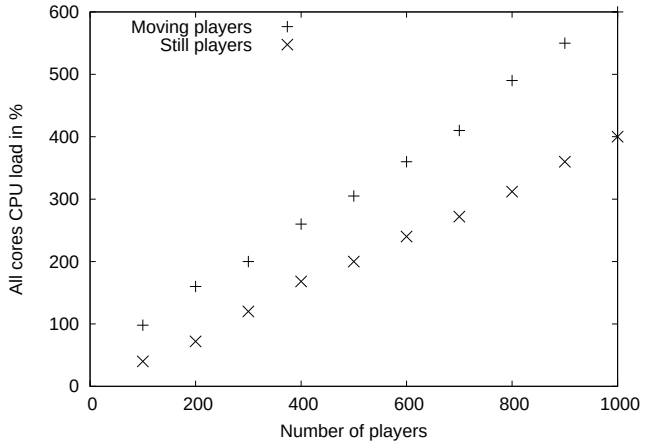


FIGURE 6.5: CPU load

6.3.3 Conclusion

One of the prominent outcomes of this experiment is that even if the map complexity may lower the maximum number of simultaneous connected players, it is not the major cause of this limitation. With an increasing number of players, the Minecraft server runs out of resources well before ten thousands.

Another lesson is that there is not a unique bottleneck. When the number of players grows, all resources become exhausted for roughly the same small number of users. Therefore, the mere presence of player entities is consuming vast resources.

Finally, the only way to go far beyond ten thousand players –the actual observed limit– together in a Minecraft map is to distribute the load generated by the entities among many machines.

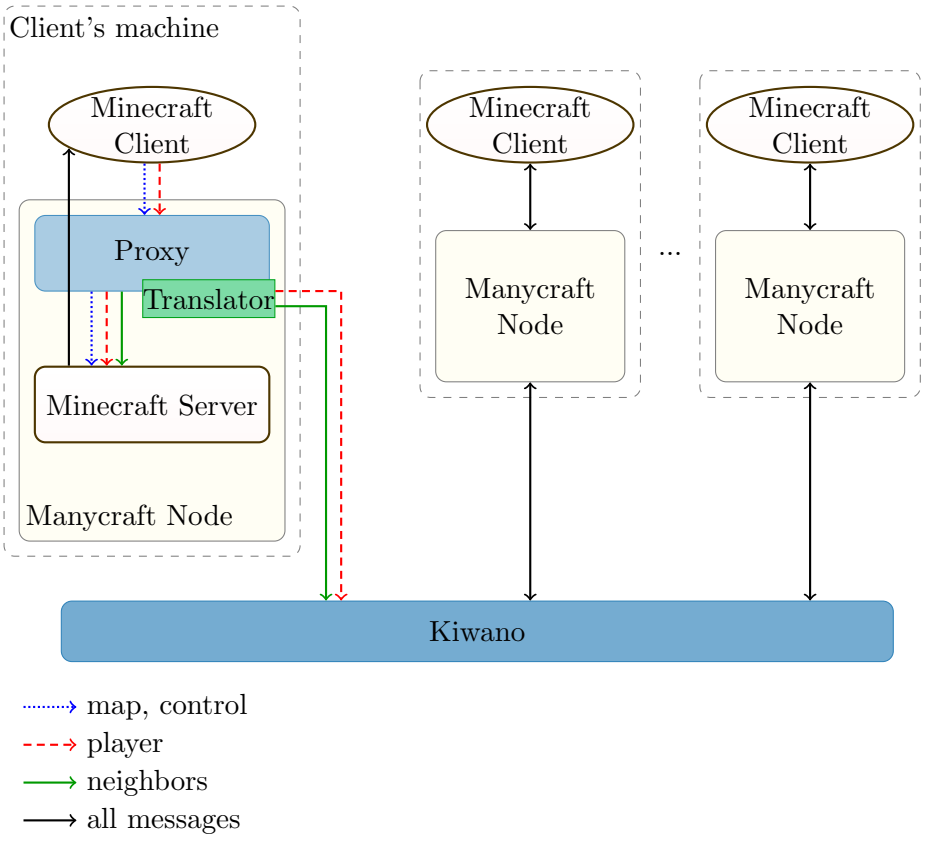


FIGURE 6.6: Manycraft node and architecture

6.4 Architecture

To do so, we propose *Manycraft* [DKV13, VDK13], our distributed architecture, meant to allow an unlimited number of players to interact in a static Minecraft map.

In our architecture the world is maintained in a distributed manner across several nodes, one for each player. A Manycraft node holds and delivers the world to the corresponding client and, to do this, it holds the static map and the neighboring entities only.

Out of the three virtual world components –terrain, objects, and avatars– in Manycraft the game state is modified only by the entities. Therefore, nodes need to communicate just via Kiwano, in order to transmit position updates and retrieve the neighborhood updates.

Granted, in our architecture nodes have a preloaded static map and host one player and its neighbors as NPCs. As we will see, the consumed resources –in terms of memory, bandwidth, and CPU– are just a fraction of what a standard populated server needs. This allows us to comfortably place the Manycraft node on the client's machine, as depicted in Figure 6.6.

6.4.1 How it works

To join the system, the player runs a Manycraft node on her own computer. The Manycraft node is a Minecraft server 'modded' with Bukkit in order to divert the entity messages to Kiwano which, in return, notifies about the events occurring in the player's neighborhood. The local Minecraft server has the static map preloaded and its main duty is to deliver the map to a unique player. The load generated by the avatars has been shifted entirely to Kiwano.

As a reminder, Kiwano [DK13] is a scalable distributed infrastructure for virtual worlds, designed to support an unlimited number of moving objects updating their position at arbitrary high frequencies.

To scale a virtual world –or an MMOG to be more precise– clients connect to the Kiwano API [22] and send three types of commands: *update* entity state, *message to all* and *private message*. On the other side, they receive four types of notifications: the full list of *neighbors*, neighborhood *updates*, *message* from another client, and *remove* neighboring entities. Plus, all Kiwano messages are encoded in json, common now in most programming languages.

Kiwano has been designed independently of any particular system and messages have an *appdata* field to carry data specific to the virtual world to scale. These messages are exemplified in Figures 6.7, 6.8, and 6.9.

```
{
  method:"update",
  urlid:"http://manycraft.net/u6nMnT",
  iid:"john.smith",
  lng:52.5, lat:15.3, angle:40, alt:40,
  appdata:{
    manycraft: {
      sneaking:false,
      pitch:0.147
    }
  }
}
```

FIGURE 6.7: Update command to Kiwano

6.4.2 Manycraft Node

To join a Manycraft world, the player downloads the node preloaded with the corresponding Minecraft static map. The node is composed of a regular Minecraft server and a *proxy*[2]. Once the node is running on their machine, the Minecraft client connects to

[2]Manycraft proxy is different from the Kiwano proxy node.

```
{
  method:"updates",
  urlid:"http://manycraft.net/u6nMnT",
  iid:"john.smith",
  new:
  [
  {
    urlid:"http://manycraft.net/w28QPu",
    iid:"alice.smith",
    lng:52.5, lat:15.3, angle:40, alt:40,
    appdata:{
      manycraft:{
        sneaking:true,
        pitch:3.2
      }
    }
  }
  ]
  old:
  [
    ("http://manycraft.net/w28QPu", "charlie.doe"),
  ]
}
```

FIGURE 6.8: Neighborhood update notification from Kiwano

```
{
  method:"msg",
  urlid:"http://manycraft.net/u6nMnT",
  iid:"john.smith",
  recipients:
  [
    ("http://manycraft.net/8Qw2RJ", "alice.smith"),
  ],
  msg:"",
  appdata:{
    manycraft:{animation:"swing_arm"}
  }
}
```

FIGURE 6.9: Message notification to Kiwano

the local Minecraft server embedded in the node through the proxy. The local server receives the packets from the client and serves the blocks of the map as the player moves. The proxy also sends, after translation, player movements and actions to Kiwano.

Kiwano notifies back about the movements and actions occurring at nodes of the neighboring players. These notifications are translated and processed by the internal Minecraft server which generates the packets for the client. Remote players movements and actions are then rendered by the client.

As players are in the same map, see each other and interact, they are together in the world.

6.4.3 Bridging all nodes over Kiwano

The essential task of the Manycraft Node is to route the relevant Minecraft traffic from and to Kiwano. The proxy inspects the messages issued by the client and routes them accordingly. All the messages, including player movements, are forwarded without modification to the Minecraft server. On position updates the Minecraft server sends the surrounding blocks. Messages concerning player actions are translated and sent to Kiwano which sends back notifications about the neighbors. These messages are translated into Minecraft messages so that the neighboring entities can be placed on the map as mobs. This way, the server attached to a player has all the information corresponding to build its corresponding interest management.

Minecraft actions to Kiwano commands

Each entity related message issued by the client is intercepted by the proxy and translated to a Kiwano command, as listed in Table 6.1. The Minecraft account name is translated into a Kiwano avatar id, see the message examples in Figures 6.7 and 6.9. Player's state modification generates a Kiwano update command, player's coordinates and yaw are respectively mapped on lat, alt, lng and angle fields. Minecraft specific description of the avatar is sent in the appdata field with appid 'manycraft'. And finally, chat and player actions are translated into Kiwano message commands.

Minecraft player action	**Kiwano command**
join	connect to Kiwano
quit	disconnect from Kiwano
kicked	disconnect from Kiwano
move	update lng, lat, alt
look	update angle
toggle sneaking	update appdata
toggle flying	update appdata
toggle sprinting	update appdata
animation	message to all appdata
chat/say	message to all
chat/tell	private message

Table 6.1: Minecraft actions translation

Kiwano notifications handling

Kiwano notifications inform a node about actions and state changes of remote players in the neighborhood, and they are structured as presented in Figure 6.8. Remote players are represented in the Minecraft server as mobs. Kiwano notifications trigger movements and actions of these corresponding mobs. Upon notification of entity arrivals/departures in player's awareness area, the mobs are created/removed accordingly in the Minecraft server. Table 6.2 sums up how Kiwano notifications are handled by the proxy.

Kiwano notification	**Minecraft handling**
update	create or update entities
remove	remove corresponding entities
message	issue corresponding action (animate an entity's arm, display message, etc.)

TABLE 6.2: Kiwano notifications translation

6.4.4 Manycraft scalability and performance

The Minecraft server embedded in the node supports only the load of one client. The traffic with Kiwano is the subset of the normal Minecraft traffic corresponding to entity updates and remains low. Finally, the translation and routing processes of the proxy do not generate any significant load.

Compared to a regular Minecraft client, the CPU load of the user's computer is only increased by the low activity of the embedded server. The memory depends on the size of the preloaded map but remains constant and low in our tests. Because in average the number of neighbors provided by Kiwano is constant, resources used by the proxy remain constant. The external communication of a Manycraft node happens only with Kiwano and the throughput corresponds only to the entity-related messages. There are no map or protocol control related messages. All in all, compared to classical setting –a Minecraft client connected to a remote multiplayer server– the load of the user's machine is lower regarding bandwidth and slightly higher regarding CPU and memory usage.

However, the number of players that can connect together is now independent of the game server. As Kiwano was shown to scale, we can guarantee the same for Manycraft.

FIGURE 6.10: Our team in Manycraft

6.5 Implementation and demo

We argued in Section 6.4 that Minecraft multiplayer mode is interesting from two aspects. Firstly, popular gaming experiences in player-versus-player and adventure mode such as racing, shooting, role playing, emphasize interactions between players and the map is not meant to be modifiable. Secondly, the creative mode, in which users build things, in order to share them, they either host a server on their own computer and allow others to connect, or record a video and share it on the Internet. The first solution is cumbersome and costly: the user needs to maintain a server for a few friends that might visit once in a while, and the second one does not offer those interested the possibility to actually visit the world. While usually played offline, this mode can greatly benefit by allowing anyone to connect and see the creations, and requires only a simple user interaction.

In our team, the implementation has been done using CraftBukkit [11] which is a Minecraft server modded to accept plugins. It is transparent to the clients, they connect and communicate with a CraftBukkit server in the same way they do with a regular Minecraft server. Plugins can listen for specific events such as players joining/leaving, avatar animations, map state changes etc., and add custom handlers. In particular, this enables intercepting messages between client and server and altering the server's response.

The Manycraft node is a CraftBukkit server with a specific plugin. In order to intercept messages to the server, the Manycraft plugin listens for events related to players and

generates the corresponding Kiwano commands, as described in Table 6.1. For example, when a player joins the server, the plugin opens a Kiwano stream. On player movement or stance modification, the plugin sends Kiwano commands to notify her neighbors.

On the other side, the plugin converts Kiwano notifications into local Minecraft events, see Table 6.2. For instance, on reception of entity updates, the plugin updates the local representation of remote players or creates it if does not have them already represented.

According to the nature of the map and different world physics, there are other types of events that can be handled to be synchronized across Manycraft nodes with the help of Kiwano. A common example is the opening and closing of doors. But these events are highly restricted by the locality in space and time. They must either not be persistent or to happen periodically, with a high frequency.

The Manycraft implementation developed within our team has been tested by our friends and colleagues invited to connect and comment about their experience. It was displayed at NetGames demo session [VDK13] and used in an MMVE virtual seminar meeting to connect several participants across the globe.

6.6 Summary

In this chapter we proposed Manycraft, a solution to allow *many* players to mine*craft* together in the same map. It is a solution designed to allow an unlimited number of players to simultaneously interact in the same static Minecraft map. Manycraft relies on the separation of the load generated by the entities from the map and control.

We forward entity related messages to Kiwano, a general architecture to scale virtual worlds, and keep on player's machine only the relevant subset of neighbors.

We are now able to scale the number of players together in the same map, and our first thought was to allow them to modify and to perceive a dynamic shared environment.

In real life, the question whether the falling tree produces a noise relates to an observer. The question that opened this chapter could also be reformulated as: Are the trees in the forest observers of the event? Could a deaf person be considered an observer for the sound of the falling tree?

This problem arises with Manycraft, or any possible architecture where the game state is maintained by the players. In our architecture the world is simulated for each player, and therefore, trees in an unpopulated forest are not simulated. The place is considered empty even when there could be non player characters. Since they do not receive

information about the world, they can be thought of as deaf persons. Actually, in this setting NPCs will not even be simulated.

In games, an observer corresponds to a player. And, thus, events that have a broad or distant effect are not currently taken in charge by our solution.

Besides taking in charge a broader range of events, future work may also include the migration of the Manycraft node to data centers, relieving the user of cumbersome installs. As nodes are not saturated by a single user's neighborhood, we envisage maintaining the world for several players within a node.

This first experiment with Minecraft suggests that a similar approach could be successfully applied to any online virtual world that allows to inspect and divert packages. In the following chapter we will present our plans to apply a similar approach to Second Life, a complex virtual world that raises new challenges.

With Manycraft a map can be populated by an unlimited number of players opening new perspectives in massive social interactions such as concerts, fairs, demonstrations, battles, etc. The first version, available at http://manycraft.net, supports static maps and simple user interactions, such as chat, pokes, and violent murders. Ongoing works focus on map dynamics and on extending player interactions.

Chapter 7

Virtual World Case Study: OneSim

Contents

> *He is not seeing real people, of course. [...] The people are pieces of software called avatars. They are the audiovisual bodies that people use to communicate with each other in the Metaverse.*
>
> Neal Stephenson – Snow Crash, 1992

We designed Kiwano independently of a particular virtual word or application. And one of the biggest challenges today is to scale popular virtual worlds that are already known to have a scalability problem such as Second Life [32, 107, 110]. Neal Stephenson envisioned in his science fiction book, *Snow Crash* [103] such a Metaverse, that allows an unlimited number of users independently of their density and distribution.

We have seen so far that once we can isolate avatar movements in a virtual world's state change we are –at least theoretically– able to manage them with Kiwano. But how practically feasible is to modify a Second Life world in order to scale? For this, we rely on an open source implementation for virtual worlds *à la* Second Life called OpenSimulator, or shortly OpenSim [29].

We use a similar approach as for Manycraft, presented in the previous chapter, in which we intercept messages from the server and divert those that change the game state. By transferring the load generated by the moving avatars to Kiwano, we have designed OneSim [DK14], a distributed system conceived to allow an unlimited number of users to be together in one contiguous region. Each user runs an OpenSim instance of the same region which is populated with their respective neighboring avatars provided by Kiwano.

This chapter begins by describing virtual worlds as precursors of the envisioned Metaverse. We mention why Second Life is different from other games and why it is worth a case study. We continue by inspecting in Section 7.2 OpenSimulator, its aspects and limitations. We are then ready to introduce our OneSim architecture in Section 7.3. We conclude this chapter with our short term vision on prospective virtual worlds and how we place Kiwano and OneSim in this context.

7.1 (Second) Life is not a game

Second Life virtual world, with a million active users logging on and inhabiting the world every month and 13,000 new ones every day [104], is still popular eleven years after its birth. Second Life is a 3D online virtual world where avatars want to do things people do in real life: Buy and sell items, play and listen to music, attend conferences, flirt and have sex.

Second Life has often been assimilated with role playing games, like World of Warcraft [37]. At Linden Lab they say it is not a game because "there is no manufactured conflict, no set objective,[...] it's an entirely open-ended experience". The environment, with its appearance, rules and objects, is created by its users together. As for Minecraft, we claim that its success is largely due to the possibility to create objects and produce a different environment. It's about avatars creating the world they inhabit.

But this is not all, as it is the case for many life simulation games. Let's take for instance The Sims which is, without objection, called a game. It is a simulation of the daily activities of some persons, and the player can make decisions about how they spend their time and how they develop skills.

But in Second Life players identify with their avatars and they are aware that behind other avatars there are other people too. This is what differentiates their behavior from the behavior of a game character. We expect them to act naturally, to move naturally and to do things people *want* to do.

Users are represented by *avatars* (also called *residents*) which are placed in the two dimensional world, also named *grid*. There are several grids, each composed of *regions*, the unit of space division. Typically, it is said that regions are maintained by the *simulators* –the processes running the world's physics and event handling, etc.– which run on *sims* –that is, the actual physical machines.[1] Each simulator runs on a single dedicated server core, which makes interest management near borders a problem.

Second Life is based on a client-server architecture, but its design is remarkably more elaborated. To enter the world, the *viewer*, that is, the client, maintains the connection to one of the several thousands running *simulators* for as long as the avatar is in that region. The avatar can walk or fly to another region in the grid in which case the viewer ends the connection to the previous and connects to the new server.

Both Second Life and OpenSimulator add to their architectures inter-operating components such as databases, services, and communication infrastructures. Shared data is accessed through services, for instance the Asset service maintains objects representations, textures, scripts, the Grid service handles the positioning of regions in the grid.

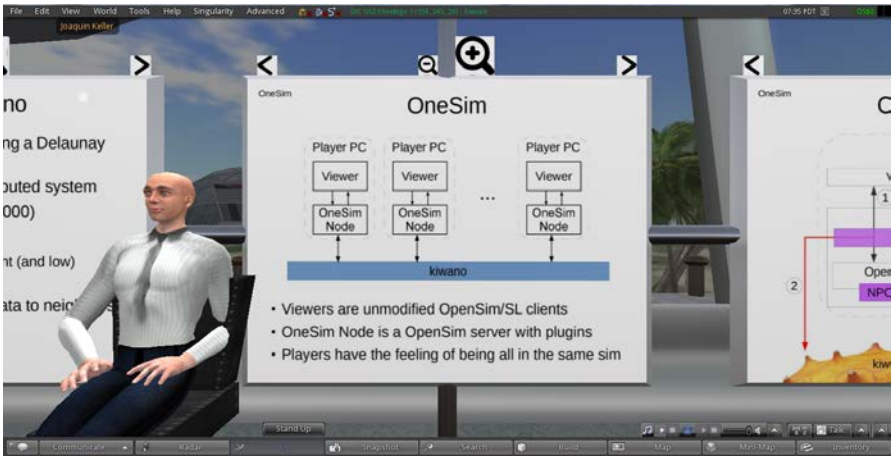
Despite a vibrant research community and an exponential growth in hardware performance, the per-region scalability has remained low with a maximum of a few hundred users [65] at best in a single region. This is the problem we want to solve by designing a distributed architecture on top of Kiwano. With OneSim, the load generated by the avatars is passed to Kiwano, which is independent of the space division and ensures average constant complexity independently of the number of avatars in the world.

7.2 Scalability study using OpenSim

In order to connect the virtual world to Kiwano, we need to have access on the sever side, to intercept avatar messages and to manipulate non-player characters. OpenSimulator [29] is an open source server to host virtual worlds which can be accessed through a variety of clients and protocols, including the ones for Second Life [29]. Independent regions can be created and populated with users' avatars for as much as the hosting server can support.

[1] A sim is a machine that runs simulators, which are the computer processes that run regions.

(A) The conference room



(B) Poster presentation

FIGURE 7.1: OneSim at MMVE Virtual Seminar. The conference room is hosted on an OpenSim server at University of California at Irvine

The viewer sends the avatar intended actions and movements, then the sim applies physical rules and computes the interactions with the elements of the décor and the other avatars before replying to everyone the actual action or movement. The number of avatars that can be hosted by a sim is limited by the cost of the simulation. The maximum number of avatars on a region can be 100, but best practice is to limit each region to 50 [32].

7.2.1 Protocol messages

Second Life protocol currently uses 568 different message types in the communication between servers and clients. It used 473 message types in 2008 and their number grows as new features are implemented. Some [109] have identified, as for Minecraft, the three main packet types: control (client connection status and security control), region (region appearance and objects in the client's interest area) and avatar (*e.g.*, position and body

appearance, chat messages exchanged with neighbors). However, we claim that these categories are not so well delimited like they were previously, for Minecraft. As we exemplify for AgentUpdate and ImprovedTerseObjectUpdate, see Figures 7.3 and 7.5, the same message may contain information relevant for multiple such categories.

The scalability problem has been attributed to the large number of events that need to be treated by the sim and the client. The Second Life protocol highlights the different processing priorities and requirements for events. Messages about user inventory allow a delay of 500ms with negligible impact on the user's experience, but they must be delivered reliably, because objects should not disappear [32, 56]. However, position updates must be delivered rapidly to the neighbors. Because if one message is lost then the next in a short delay will replace it anyway, with no noticeable impact on users' experience.

7.2.2 The Hypergrid, a scalable web of regions

OpenSim already identifies and separates some concerns. There are several specialized services (*e.g.*, login, user, grid, asset, inventory) that maintain user data persistent for subsequent sessions. Intended to provide a coherent description across the simulators in a grid, they also provide a certain scalability, because the treatment of some tasks is shifted to a dedicated entity.

In order to connect avatars across virtual worlds, OpenSim offers an optional facility, the Hypergrid [91], to allow users to visit other OpenSimulator grids across the web from their 'home' OpenSimulator instance.

From an architectural point of view, it is a federation architecture and protocol for Second Life and OpenSimulator virtual worlds that supports the seamless transfer of user agents and assets between them. The Hypergrid has been envisaged collaborative and interoperable, as precursor of the Metaverse [91].

It's worth noting that the scalability in Hypergrid relies on an important architectural feature of OpenSim, namely, the separation of various aspects of the virtual world, such as the login service, assets, inventory, and others. As each has specific requirements, each one is handled separately. Objects (items) need to be persistent, avatars able to communicate with one another and the virtual word scalable.

Furthermore, the Hypergrid and our proposed architectures enable interoperability at variable degrees. Kiwano enables everyone to be together, while a common items database enables users to share the same objects. The viewer –the designed virtual world thereof– is free to share and combine different aspects with other worlds. This

feature provided by Kiwano –and other possible APIs such as Kwery– is discussed in the next chapter, Section 8.2, about the interoperability of virtual worlds.

7.3 OneSim

Since the early 1970s, when the first multi-user graphic virtual world appeared, the algorithmic complexity of running Massively Multiuser Virtual Environments (MMVEs) is $\mathcal{O}(N^2)$, where N is the number of users together in the same region. Reducing the visibility to a smaller surrounding area [80, 83, 84] does not solve the problem: user density may vary sharply by orders of magnitude, thus making the size of the area inadequate.

Peer-to-peer solutions for Second Life [107] have addressed the scalability in the complexity of the scene –that ism the numbers of objects and avatars altogether– by trading some consistency and latency. Using Kad, the object management is distributed among the peers and each one accesses its surrounding, fixed-size area of interest. A similar approach [110] evaluates how a peer-to-peer avatar management based on Delaunay triangulations like Solipsis [83, 84] and VON [80] provides scalability for Second Life. The results reveal mostly that areas of interest are practically inadequate, and that the use of peer-to-peer overlays has additional consistency and latency costs.

With Kiwano [DK13], we have taken a radical approach: users can see and interact only with their neighbors. Kiwano's neighboring relation is designed such that the number of neighbors remains roughly constant regardless of the avatar density distribution. Thus, the complexity for computing and transmitting interactions has been dramatically reduced to $\mathcal{O}(N)$, where N is the total number of avatars. The overall complexity to maintain the neighborhood relation in Kiwano has a mean complexity of $\mathcal{O}(N)$. Otherwise said, the per-user computing cost is constant and does not grow with N. This is the first step to scalability.

7.3.1 Proposed architecture

OneSim [DK14] is a distributed architecture of nodes, each associated to a viewer. All the nodes are connected and communicate through Kiwano in an architecture similar to that of Manycraft, presented in the previous chapter. In this architecture, where each user hosts a node, there is no limit in the number of users. Let's unfold the details of the OneSim node before explaining how they communicate together.

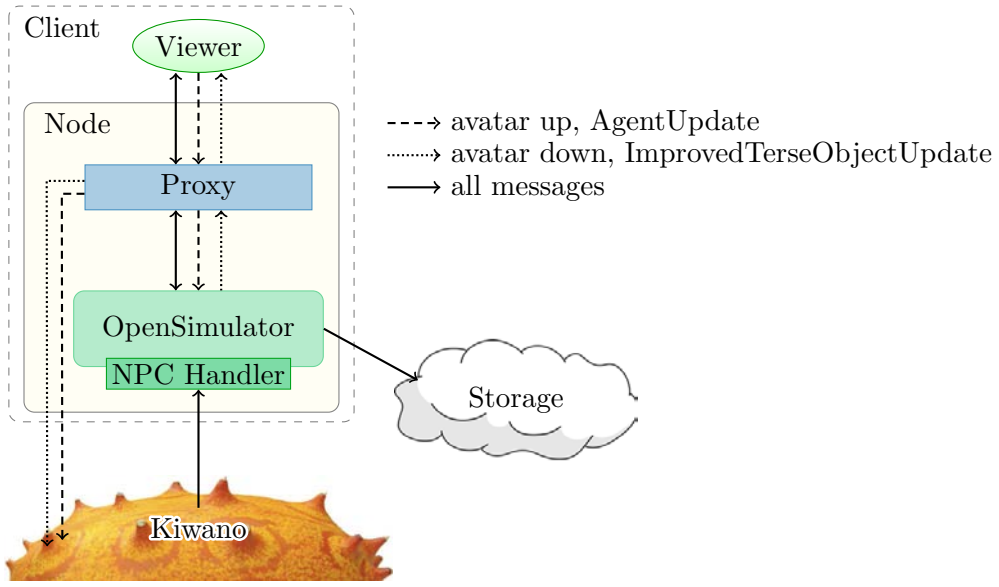


FIGURE 7.2: OneSim node

A OneSim node consists of an instance of a regular sim, holding a local copy of the region, and a proxy to forward the avatar messages to Kiwano, see Figure 7.2. In the local sim, the neighbors provided by Kiwano evolve as non-player characters (NPCs).

The packets between the viewer and the embedded sim are forwarded unmodified by the proxy. Some of these packets contain avatar updates: (1) from the viewer to the sim (*e.g.*, AgentUpdate) and (2) from the sim to the viewer (*e.g.*, ImprovedTerseObjectUpdate). The message from the viewer may carry extra data still needed for the final representation while the message sent by the sim contains the result of the simulation with the actual position and state of the avatar. They are intercepted to generate Kiwano updates and messages-to-neighbors respectively.

Messages sent to Kiwano generate notifications for all neighbors and are used to control the corresponding NPC avatar in each neighbor's viewer. In addition to position updates, these messages also bear avatar animation, orientation, appearance, chat and other avatar actions. We remind the reader that internally, Kiwano passes the messages containing these extra features at the proxy level.

7.3.2 Implementation with Kiwano

To be able to extract and to process avatar positions the most convenient is to intercept position update messages coming from the client. These are encoded in the AgentUpdate message, listed in Figure 7.3.

```
{  AgentUpdate High 4 NotTrusted Zerocoded
   {  AgentData                Single
      {   AgentID              LLUUID  }
      {   SessionID            LLUUID  }
      {   BodyRotation         LLQuaternion    }
      {   HeadRotation         LLQuaternion    }
      {   State                U8      }
      {   CameraCenter         LLVector3       }
      {   CameraAtAxis         LLVector3       }
      {   CameraLeftAxis       LLVector3       }
      {   CameraUpAxis         LLVector3       }
      {   Far                  F32     }
      {   ControlFlags         U32     }
      {   Flags                U8      }
   }
}
```

FIGURE 7.3: AgentUpdate message structure [23]

```
{  AgentUpdate:
   {  AgentData:
      {   AgentID:             15054fa73a9f4c239603c0129ad43f5e  }
      {   SessionID:           249ace543b5340179ba8cdaa718893ce  }
      {   BodyRotation:        <  0     0     0.3409626     0.9400768  > }
      {   HeadRotation:        <  0     0     0.2362511     0.971692   > }
      {   State:               0                                        }
      {   CameraCenter:        4.663468      40.48997     95.06865     }
      {   CameraAtAxis:        0.7674317     0.6410149    0.01217942   }
      {   CameraLeftAxis:      -0.6410625    0.7674887    0            }
      {   CameraUpAxis:        -0.009347567  -0.00780777  0.9999258    }
      {   Far:                 64             }
      {   ControlFlags:        33566736       }
      {   Flags:               0              }
   }
}
```

FIGURE 7.4: AgentUpdate message example [23]

A user and a session between a viewer and a simulator are uniquely identified by AgentID and SessionID respectively. The avatar's desired body and head rotation are transmitted to the sim encoded in the fields BodyRotation and HeadRotation. Information for interest management is according to the camera position and the viewing distance.

Most importantly for our study, the desired direction of movement is encoded, somewhat obscurely, in the Flags field. This encodes the differences to the previous position as indicated by the user's input and can be none, forward, backward, left, right, up or down [32, 56].

Figure 7.4 shows an example of such a message were the transmitted values are differences from the previous camera position. Since the Flags field remains null, we see that the avatar's position did not change since the previous message.

All this information is processed by the simulator which decides whether the indicated movement is legitimate. Then, it informs all other observing clients, including the user itself, about the new position.

```
{  ImprovedTerseObjectUpdate High Trusted Unencoded
   {  RegionData                Single
      {  RegionHandle           U64     }
      {  TimeDilation           U16     }
   }
   {  ObjectData                Variable
      {  Data                   Variable  1     }
      {  TextureEntry           Variable  2     }
   }
}
```

FIGURE 7.5: ImprovedTerseObjectUpdate message structure [23]

```
{  ImprovedTerseObjectUpdate:
   {  RegionData:
      {  RegionHandle:  1089616023372032 }
      {  TimeDilation:  65132            }
   }
   {  ObjectData:
      {  Data:          1a b0 02 00 00 00 23 45 73 43 1c 1a 49 43 a9 d7 }
      {  Data:          ce 41 ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f ff 7f }
      {  Data:          ff 7f ff 7f ff ff ff 7f ff 7f ff 7f     }
   }
}
```

FIGURE 7.6: ImprovedTerseObjectUpdate message example [23]

The simulator's authoritative decision and the unique game state avoid cheating and, more importantly, inconsistencies among multiple users' views. Of utmost importance is yet, the mediation of non-commutative actions. To resume our example with Alice and Bob, when Alice and Bob shoot each other simultaneously in the virtual world, the simulator decides a total order on the received events, and thus, their views will eventually converge to the same game state. According to the server, the first one to have sent the shooting action will survive.

Finally, changes in the game state are transmitted from the sim to the clients with the ImprovedTerseObjectUpdate message. Avatar position is encoded in the Data field, as listed in Figure 7.5.

In OneSim the node is responsible to intercept these messages containing avatar updates and to translate them into Kiwano messages. For example the AgentUpdate message in Figure 7.4 becomes translated into the one in Figure 7.8. In return, incoming neighborhood updates from Kiwano as in Figure 7.9 are used to update neighbors' NPCs in the embedded server. The local server finally informs the attached viewer about all visible changes with messages of the type ImprovedTerseObjectUpdate, as in Figure 7.6.

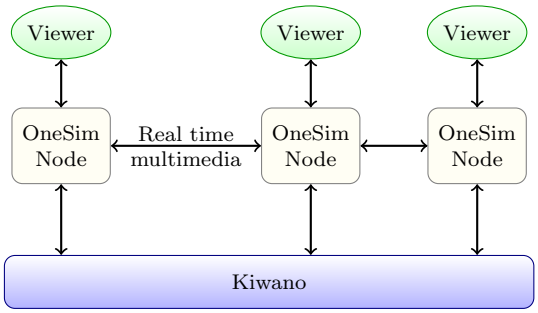


FIGURE 7.7: OneSim architecture

7.3.3 Other shared data

To enable voice, OpenSimulator uses several third party solutions, Mumble and Vivox are two popular examples. They employ a dedicated application server per region and they broadcast the voice to all avatars in that region.

However, they need special configuration/installation well in advance. Multiple instances of the same region require each a dedicated voice server. Plus, having nodes that constantly leave and join the system, makes the deployment of one remote server for each node impractical.

That's why we propose to broadcast voice and other real time (streaming) data to the neighboring avatars using a Mumble server for each sim. Voice and other streamed flows of data can be handled by each node and sent only to the neighbors' nodes. Moreover, having already the information about the map and the distance between the avatars, the node is able to morph a spatial sound.

For sharing the same region map between all nodes, we can use an archiving feature. The OpenSimulator Archive (OAR) function saves an entire region's content to a single file so that it can be later reloaded by any other sim. Also, the Inventory Archive (IAR) function saves the necessary content to a single file so that items including textures, sounds, scripts and objects can be later reloaded [29].

7.3.4 Scalability and limitations

As each node hosts a simulation of the world this may lead to inconsistencies. This is why in OneSim we propose that every non-static object has a master sim. For instance, the avatar and the inventory of a user have their reference state and position computed by the associated sim. Remote sims may compute different states but the master sim view prevails and is propagated to all others. This way, inconsistencies do no last. As sometimes the state computed by a local simulator differs from what the master

```
{
  method:"update",
  urlid:"opensim/15054fa73a9f4c239603c0129ad43f5e",
  iid:"249ace543b5340179ba8cdaa718893ce",
  lng:52.5, lat:15.3, angle:40, alt:40,
  appdata:{
    onesim: {
       BodyRotation:    <  0     0     0.3409626     0.9400768  >
       HeadRotation:    <  0     0     0.2362511     0.971692   >
       State:           0
       CameraCenter:    4.663468      40.48997     95.06865
       CameraAtAxis:    0.7674317     0.6410149    0.01217942
       CameraLeftAxis:  -0.6410625    0.7674887    0
       CameraUpAxis:    -0.009347567  -0.00780777  0.9999258
       Far:             64
       ControlFlags:    33566736
       Flags:           0
    }
  }
}
```

FIGURE 7.8: Update command to Kiwano

simulator dictates, the laws of physics may be temporarily not respected. Using OneSim in games with rapid pace actions might produce poor user experience.

In dense crowds, the user experience might also differ from what we have been accustomed: we can only see the neighboring avatars and not the whole crowd in the visibility area.

Today's implementation of Kiwano provides, in average, 70 neighbors. However, the number of neighbors that can be handled by a node is limited by the capacity of the embedded sim. But as NPCs consume less resources than full avatars, we can expect the total number of neighbors to be in the tens, for an end-user machine.

7.4 Summary

Second Life is still the very popular virtual world that extends to more immersive technologies such as Oculus [86]. Therefore solving the scalability problem is of utmost importance.

Indeed, many limitations in the performance of an OpenSim server come from the large number of events needed to be treated. With Kiwano, we provide three levels af neighborhood. As these events come mostly from the avatars, it would be of interest to investigate the possibility to filter these messages according to the distance in the neighborhood graph. We all the reasons to believe that this will reduce considerably the overhead on both, the server and the client, without considerable impact on user's experience.

```
{
  method:"updates",
  urlid:"onesim/",
  iid:"",
  new:
  [
  {
    urlid:"http://manycraft.net/w28QPu",
    iid:"alice.smith",
    lng:52.5, lat:15.3, angle:40, alt:40,
    appdata:{
      onesim: {
         BodyRotation:     <  0     0     0.3409626     0.9400768  >
         HeadRotation:     <  0     0     0.2362511     0.971692   >
         State:            0
         CameraCenter:     4.663468      40.48997     95.06865
         CameraAtAxis:     0.7674317     0.6410149    0.01217942
         CameraLeftAxis:   -0.6410625    0.7674887    0
         CameraUpAxis:     -0.009347567  -0.00780777  0.9999258
         Far:              64
         ControlFlags:     33566736
         Flags:            0
      }
    }
  }
  ]
  old:
  [
    ("http://manycraft.net/w28QPu", "charlie.doe"),
  ]
}
```

FIGURE 7.9: Neighborhood update notification from Kiwano

As for now, we have shown that the implementation of one node per user is practically feasible. As the resources consumed by non-player characters can be reduced, we plan to investigate the possibility to free the client from the load of the server, by hosting it remotely, in the cloud. In this way, with more powerful machines, we allow more avatars with their neighborhoods to be hosted the same OneSim node.

All in all, with Onesim we have seen how it is possible to meet an unlimited number of avatars in the same region, a borderless space, using a distributed architecture for Second Life.

The infrastructure provided by Kiwano can be employed to scale diverse virtual worlds and a similar architecture, adapted to handle the Minecraft protocol messages, have been successfully implemented in Manycraft, see Chapter 6. In its intent to contribute to a long lasting research problem, the scalability of virtual worlds, Kiwano provides a novel approach that complement the widely used region-based and shard-based solutions.

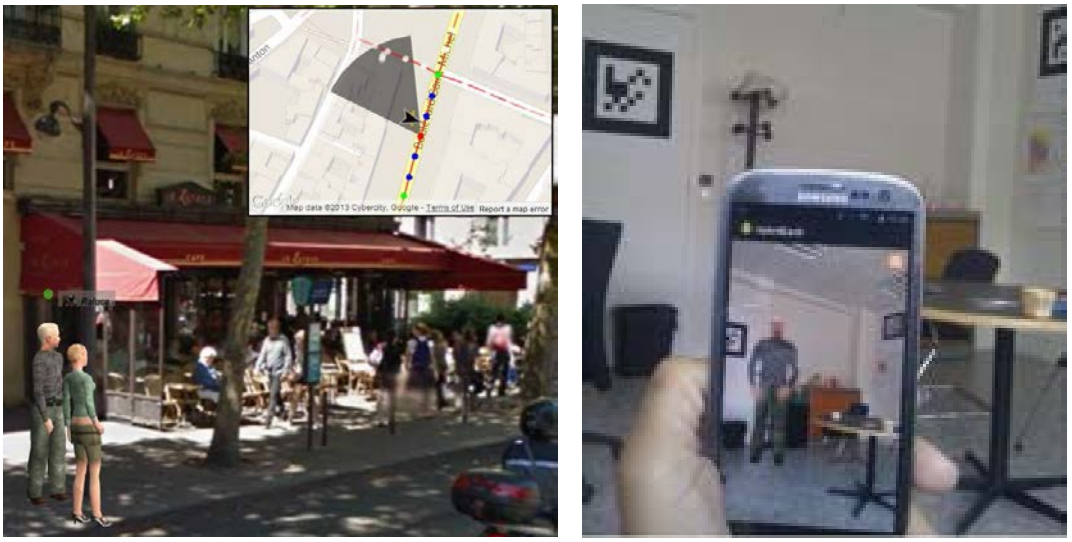
Chapter 8

Interoperability for a Shared Hybrid Reality: HybridEarth

Contents

HybridEarth [dCDKT14] is a project conceived and developed in our team, materializing the idea that avatars and people inhabit together the same world: the Earth. They see each other and interact with no difference as for which is which. That's what is called social mixed reality: a hybrid world, half real, half virtual, with avatars and people side by side in the same shared space. This world is dual in nature: it can be entered either as an avatar in a virtual world, copy of the real world, or as a physical person with an augmented reality device to perceive the virtual side [dCDKT14]. What seemed to be science fiction a few years ago is indeed becoming reality.

Sensors to scan the natural environment have significantly increased in their number and performance, and this growth continues at an exponential rate. Their use led to what we call mirror worlds, that is to say virtual worlds resembling more and more the real environment [74]. The typical example is Google Street View [35], deploying all over the planet those cars full of sensors, spherical cameras, rotating lasers for 3D scanning, antennas to map the Wifi hotspots, along with others. But there is more, people take



(A) The mirror world with Street View (B) Mobile augmented reality application

FIGURE 8.1: HybridEarth applications [dCDKT14]

panoramas, even streams of geolocated videos, with their smartphones, tag them with their location, and share them over the Internet.

Using 360° panoramic images from Google Street View and our own databases, HybridEarth implements a mirror world as static navigable imagery accessible with a web browser. It completes the virtual world by adding users' avatars to it, with an interface showed in Figure 8.1(A).

On the other hand, HybridEarth employs a geolocation technique with a precision in centimetres to insert users wearing augmented reality devices at their actual location in the virtual world. The accuracy is crucial, as it enables a seamless and coherent interaction between humans and virtual elements, including avatars.

Today's virtual worlds can support at most few thousands users together in a contiguous space. A social mixed reality world covering the whole planet should be able to host more. And for that, HybridEarth relies on Kiwano, our novel distributed system to scale virtual worlds.

HybridEarth is accessible from `http://hybridearth.net` where users can enter the world either by installing an application on their Android device or by walking their avatar in the virtual world.

We start this short chapter with a presentation of HybridEarth, explaining its two-folded architecture, the mirror world and the augmented reality. Section 8.2 shows how these two worlds are interconnected. We expand the idea to show general interoperability for virtual worlds using Kiwano and, prospectively, Kwery. We conclude this last part of the

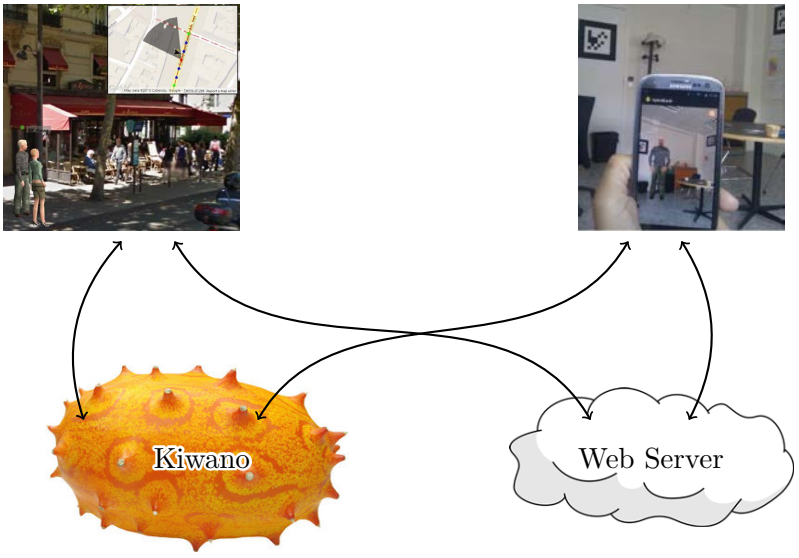


FIGURE 8.2: HybridEarth architecture

thesis with a summary of the contributions we bring with Kiwano to the interoperability of virtual worlds.

8.1 HybridEarth: Mixed reality at planet scale

To enter HybridEarth, you can choose to do it from where you are, using an augmented reality device and sharing your position [105], or by placing your alter-ego avatar in the web-based virtual world [62]. These two applications have been developed independently in our team. Users send their positions to and receive the neighbors from Kiwano. Other application data is shared via the Web Server as depicted in Figure 8.2 (we do not detail its design).

Before describing how Kiwano ensures interoperability for avatars between virtual worlds, we present in this section how these two applications enact a mixed reality world.

8.1.1 A Mirror World based on Street View

To allow people to insert their avatars, HybridEarth creates a virtual world, copy of the real world, using the Street View imagery from Google. Spherical panoramas are geolocated and interconnected in a graph structure were paths match actual pathways.

In this virtual world, each panorama corresponds to a possible position for the camera –a third person, tracking camera. In classical virtual worlds the terrain has usually a full 3D

representation, allowing the camera to be positioned at will. But in our setting, where the terrain is a navigable imagery mapped into graph structure, the camera movements are constrained to only zoom, rotate from the center, and jump from one panorama to another following the allowed paths.

Avatars populate this copy of the real world in the same way they do in classical virtual world. Avatars are animated, they walk around, they can see each other and interact via text chat or voice.

The user accesses the world through a web application where they place their avatar in the Street View, together with the neighboring avatars received from Kiwano and the world description from the Web Server. As the avatar walks and moves around, the camera follows them with the best view in the navigable imagery.

Google's Street View imagery already maps many of the outdoor locations of the planet and even indoor venues in selected places. However, many places of interest (most of Africa, Asia, and our own office for instance) are not covered. In addition, HybridEarth comes with a smartphone application to easily extend the database in a collaborative manner. With this app users –ourselves especially– take spherical pictures, locate them on the map and draw the connecting paths.

8.1.2 Augmented Reality and Geolocation

The virtual world of HybridEarth corresponds to actual places, populated with real people. Bringing everyone together consists of (1) placing in the virtual world avatars of the real people and (2) making remote people's avatars visible in the augmented reality.

Once the panoramas are precisely geolocated –that is, the camera's exact geographic coordinates are known– avatars controlled from the web interface have an accurate position relative to the camera. But for the people in the real world, wearing an augmented reality device, the position must be precise enough to make human to avatar interaction accurate and coherent for every participant.

Augmented reality has become widespread once with the devices enabling it: smartphones, tablets, eyeglasses (such as Google Glass, Laster) or other head mounted displays. All these devices sport various location sensors –GPS[1], gyroscopes, Wifi, and more importantly, cameras– and communicate over the Internet. Thus, they also make the wearer visible in their actual position.

[1] Global Positioning System and, by extension, GPS device

FIGURE 8.3: Virtual worlds interoperability with Kiwano. People and avatars from various virtual worlds can be together in the same shared space.

The current version of HybridEarth uses visual markers to accomplish two tasks: (1) to correctly place the wearer's avatar in the virtual world and (2) to place the neighboring avatars in the field of vision. For a correct interaction between humans and avatars, a precise location is necessary. The location computation is compensated with the use of GPS, Wifi, accelerometers, and gyroscopes when available.

Avatars in one's field of vision are provided by Kiwano, and other virtual elements, such as skins for instance, by the Web Server.

8.2 Interoperability for virtual worlds

With Kiwano we have shown that it is possible to scale a virtual world by orders of magnitude beyond the current state of the art. But more than that, by taking apart the treatment of avatar updates under a unified API, we can further connect everyone to the same shared space. Avatars in different virtual worlds are able to be in the world and interact. Let's begin this discussion with HybridEarth.

8.2.1 HybridEarth scalability

Today's virtual worlds barely reach hundreds, at best thousands, of simultaneously connected users in the same space. However, in this new territory: ubiquitous mixed reality, everyone should be able to get in. It means a virtual world as vast as planet Earth, with

billions of avatars. This is many orders of magnitude bigger than any existing virtual world.

By connecting to Kiwano, HybridEarth scales in the number of users. The application connects to the provided API, sends the absolute position and receives updates about the neighbors only. These are used to control neighbors' corresponding avatars.

The two HybridEarth applications, web and mobile, have been developed independently in our team, but they are just two different ways to acess the same information. Everyone is connected to the same virtual world and interacting through Kiwano.

The two applications described above share the decorum *a priori*. It has different representations –real and digital– but it is basically the same. The web application has a static decorum, provided by the navigable imagery. This is composed of static images taken well in advance, and uploaded to the database.

On the other hand, the augmented reality application does not need the terrain information. According to the device that deploys the app, the decorum is seen directly, as for augmented reality transparent glasses, or received in real time from the camera, for instance on smartphones, tablets or virtual reality head-mounted display such as Oculus glasses.

However, both types of users see the same avatars that inhabit, at the same time, the real and the virtual world.

8.2.2 Efficiency and interoperability for all virtual worlds

To scale Manycraft using the same Kiwano infrastructure, what we did was to scale and project the flat surface of the terrain on a small enough portion of the sphere in Winterfell. Of course, in terms of neighborhood the differences would be hardly noticeable.

Kiwano makes abstraction of the application connected. Internally, there is no difference between avatars from Manycraft and HybridEarth. Therefore, walking their HybridEarth avatar in Winterfell, one can see the avatars of Manycraft players and vice-versa, like in Figure 8.4. Definitely, by separating the treatment of avatar updates we were able to achieve not only massive scalability, but also virtual world interoperability.

Of course, seeing undesired avatars can be avoided by carefully mapping the each virtual world's terrain surface onto the sphere. Somewhere in the middle of Pacific, with no human inhabitants and no Street View, should be appropriate for this setting.

(A) HybridEarth (B) Manycraft

FIGURE 8.4: Avatars in HybridEarth and Minecraft interacting through Kiwano

In actual systems most regions are empty while 1% are overloaded [114]. This is highly inefficient as the resources allocated for the empty regions remain unused. But also, when they do have participants, virtual worlds are too limited. In Second Life meetings gather less than a hundred simultaneous avatars in the same region. Using Kiwano, the world is available while no additional resources are consumed when no avatar is connected. Moreover, large groups and high densities become possible.

8.3 Summary

HybridEarth is a mixed reality world: it means that it can be entered either from a desktop as a classical virtual world; or using an augmented reality device to see the surrounding virtual elements.

To summarize HybridEarth, it was conceived to demonstrate several technology advances developed within our team:

- A virtual world based on spherical panoramas accessible with a web browser;
- A dual use of augmented reality markers to visualize virtual content and geolocate with a precision in the centimeter range;
- Kiwano, a scalable distributed infrastructure for virtual worlds.

And using Kiwano, we achieved much further, we proved the interoperability of multiple –virtual and real– worlds:

- The real world connected through geolocation augmented reality devices;
- A mirror world based Google Street View and capable to extend to any desired location on Earth;

- Existing virtual worlds, with a case study on Second Life (OneSim prospective);
- Maps of popular online game Minecraft (Manycraft).

In this manner, running the same Kiwano infrastructure for all connected virtual worlds is cost-efficient. We were able to use the same resource concurrently during the development of Manycraft and HybridEarth.

Using the same approach with Kwery we can make virtual worlds share dynamic objects in a uniform manner.

Chapter 9

A 3 Point Perspective

Contents

Solving the scalability problem of virtual worlds is difficult and it is often seen as a future optimization. On one hand, few users will not motivate a scalable extension. On the other, the initial exponential growth does not leave enough time to do this.

In this thesis we provided a solution to support theoretically an unlimited number of avatars in the same virtual space and tested it with tens of thousands of moving avatars, well beyond the current state of the art. We built our solution independently of the virtual world to scale and backed it up with designs for three different scenarios: a new and popular MMOG (Minecraft), a well established virtual world (Second Life), and a mixed reality world (HybridEarth).

9.1 Synthesis of the contribution

Our scalability analyze shows that current centralized implementations are far from the requirements needed to run virtual worlds in terms of numbers of avatars (and moving objects) and the frequency of updates supported.

To enable scalability in a contiguous virtual space, we handle separately the main sources of load namely, avatars and moving objects. Our goal is therefore to provide independent, specific solutions to scalability.

To allow an unlimited number of avatars to simultaneously evolve and interact in a contiguous virtual space, we proposed Kiwano. In Kiwano we employ the Delaunay triangulation to provide each avatar with a constant number of neighbors independently of their density or distribution. The avatar-to-avatar interactions and related computations are then bounded, allowing the system to scale. The load is constantly balanced among Kiwano's nodes which adapt and take in charge sets of avatars according to their geographic proximity. We evaluated our implementation with tens of thousands of avatars connecting to a Kiwano instance running across several data centers, outperforming by many orders of magnitude the current state of the art.

This implementation and its public API have been used for virtual worlds design and developments, *i.e.*, Manycraft, OneSim, and HybridEarth.

Furthermore, we proposed Kwery, a distributed spatial index to perform efficient reverse geolocation queries on large numbers of moving objects updating their position at arbitrary high frequencies. Spatial queries are forwarded only to some nodes, those that have their zone intersecting the query zone. In Kwery, each node maintains a zone which is the minimum bounding rectangle encompassing the group of hosted objects selected in spatial proximity. The resulting load is distributed in a self-adaptive tree structure.

Using resources independently of their location, our systems are, to our knowledge, the first cloud infrastructures for virtual worlds capable to provide massively distributed and self-adaptive solutions with unbounded scalability.

9.2 Future work

Kiwano demonstrates good load balancing and responsiveness with usual avatar mobility. At this moment, for some frequency of updates or amplitude of movements the performance of Kiwano will slowly deteriorate. For that matter, it remains an open problem how to efficiently treat special avatar behaviors such as mass teleportations or flocking. Future tests need to include the effect on the system's performance for drastic variations in density.

Kiwano can benefit from filtering mechanisms and we estimate that a wisely designed filter can lower significantly the average frequency of messages needed to be sent to the nodes. In social scenarios the vast majority of movements will not change the visible neighbors.

Kiwano has been used to scale up static worlds and works well for social interaction. But it does not cover all situations. For instance, neither it handles non-commutative

operations –*e.g.*, if Alice and Bob shoot each other, the order of execution matters– nor provides synchronization for non-avatar objects, whose master replicas have to be hosted elsewhere.

We plan to do this in Manycraft. Ongoing works focus on extending player interactions and providing mutable environment. This is a challenging task taking in consideration that the world has an instance for each player. Besides that, future work may also include the migration of the Manycraft node to data centers, relieving the user of cumbersome installs. The performance of Minecraft makes that nodes are not saturated by a single user's neighborhood, and thus, we envisage to employ a node to maintain several players.

Kwery is an ongoing project that already showed us that scalability for timely reverse geocoding in virtual worlds is feasible. Kwery has been tested using R-trees, but makes no assumption on the nature of the local indexes. Zone nodes may vary their local indexing structure. And this makes Kwery a scalable, distributed architecture for any incremental spatial index with support for range queries.

9.3 General perspectives

We based our scalable solutions on a separation of concerns: We handled avatars and moving objects apart from the static virtual world terrain. In this way, we were able to (1) break the huge scalability problem into smaller and easier ones so that (2) we came with specialized novel solutions. The geographic independence of our solutions allows us to provide scalability regardless of the distribution and the density.

In Kiwano we define a new neighborhood relation for avatars based on Delaunay triangulation graphs. The number of neighbors is bounded, thus enabling linear complexity and a distributable data structure. As long as the degree is bounded, the neighborhood graph can be chosen arbitrarily.

Basically, we now perform an incremental distributed computation of the Delaunay triangulation. It would also be interesting to offer it as a fast and distributed solution for computational geometry applications.

The Delaunay graph offers a symmetric relation and can be coupled with another symmetric relation, for instance a fixed distance. For now, the Delaunay graph provides three levels of neighborhood. But as we tend to concentrate our attention mostly on the nearby environment, we can employ the level in the relation coupled to reduce the number of events. For instance, only close avatars and objects need a fully detailed representation, while those far away may be represented as silhouettes.

Using Kwery we also described how to implement a publish-subscribe system and show how to extend our solutions for more general queries over multiple attributes.

With Kiwano, HybridEarth demonstrates the concept of hybrid reality by enacting interoperability of real and virtual world representations. Such social mixed reality world can greatly benefit by integrating new technologies and extending the range of interoperability.

Cloud solutions are progressively being adopted and are already demonstrating their benefits. With Kiwano and Kwery we are part of this improvement. Yet, their use is still limited and research and industry look towards new cloud-based solutions. Of course, these systems can greatly benefit from the important research in distributed algorithms and peer-to-peer architectures.

Bibliography

[10] Bing maps. www.bing.com/maps. Retrieved November 10, 2014.

[11] Bukkit, a free, opensource, extension of minecraft multiplayer server. http://bukkit.org. Retrieved August 10, 2013.

[12] Computational geometry algorithms library. http://www.cgal.org. Retrieved November 3, 2014.

[13] Counter-strike: Global offensive. www.counter-strike.net. Retrieved November 10, 2014.

[14] A directory of minecraft servers. http://minecraftservers.org. Retrieved August 10, 2013.

[15] Geocaching - the official global gps cache hunt site. www.geocaching.com. Retrieved November 20, 2014.

[16] Gnu triangulated surface library. http://gts.sourceforge.net. Retrieved November 3, 2014.

[17] Google earth. earth.google.com. Retrieved November 10, 2014.

[18] Hybrid earth blog. http://blog.hybridearth.net.

[19] Hybrid earth: Mixed reality at planet scale. http://hybridearth.net.

[20] Improving your minecraft server's performance. http://sk89q.com/2013/03/improving-your-minecraft-servers-performance. Retrieved August 20, 2013.

[21] Ingress. www.ingress.com. Retrieved November 20, 2014.

[22] Kiwano api. http://kiwano.li.

[23] libopenmetaverse. http://lib.openmetaverse.org. Retrieved October 30, 2014.

[24] Libspatialindex documentation. http://libspatialindex.github.io/. Retrieved November 3, 2014.

[25] Manycraft project. http://manycraft.net. Retrieved August 10, 2013.

[26] Minecraft current stable protocol. http://mc.kev009.com/Protocol. Retrieved August 20, 2013.

[27] Minecraft flatlands. http://minecraftwiki.net/wiki/Superflat. Retrieved August 20, 2013.

[28] Minecraft official website. https://minecraft.net. Retrieved October 6, 2014.

[29] Opensimulator. http://opensimulator.org. Retrieved September 3, 2014.

[30] Postgis. http://postgis.refractions.net. Retrieved November 3, 2014.

[31] Quake iii arena. www.quake3arena.com. Retrieved September 3, 2011.

[32] Second life wiki. http://wiki.secondlife.com/. Retrieved October 6, 2014.

[33] Tinder. www.gotinder.com. Retrieved November 20, 2014.

[34] Ultima online. www.uo.com. Retrieved November 10, 2014.

[35] What is the google maps api? https://developers.google.com/maps/. Retrieved November 3, 2014.

[36] When to start thinking about scalability? - programmers stack exchange. http://programmers.stackexchange.com/questions/203928/when-to-start-thinking-about-scalability. Retrieved November 10, 2014.

[37] World of warcraft - battle.net. http://battle.net/wow. Retrieved November 10, 2014.

[38] David Almroth. Pikko server. In *Erlang User Conference*, 2010.

[39] Evangelos Bampas, Anissa Lamani, Franck Petit, and Mathieu Valero. Self-stabilizing balancing algorithm for containment-based trees. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 191–205, 2013.

[40] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[41] Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, and Etienne Riviere. Voronet: A scalable object network based on voronoi tessellations. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.

[42] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[43] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB), Mumbai, September 3-6, India*, 1996.

[44] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 353–366, 2004.

[45] Ashwin R. Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), Seattle, WA, USA, August 17-22*, pages 389–400, 2008.

[46] Ashwin R. Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *3rd Symposium on Networked Systems Design and Implementation (NSDI), San Jose, California, USA, May 8-10, Proceedings.*, 2006.

[47] Silvia Bianchi, Pascal Felber, and Maria Gradinariu Potop-Butucaru. Stabilizing distributed r-trees for peer-to-peer content routing. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1175–1187, 2010.

[48] Jean-Daniel Boissonnat, Monique Teillaud, Olivier Devillers, and Mariette Yvinec. Triangulations in CGAL. In *16th annual Symposium on Computational Geometry*, number 21957, pages 5–19. ACM, 2000.

[49] Bruno Van Den Bossche, Bart De Vleeschauwer, Tom Verdickt, Filip De Turck, Bart Dhoedt, and Piet Demeester. Autonomic microcell assignment in massively distributed online virtual environments. *Journal of Network and Computer Applications*, 32(6):1242–1256, 2009.

[50] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NETGAMES*, page 6, 2006.

[51] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, page 7, 2000.

[52] Eliya Buyukkaya, Maha Abdallah, and Romain Cavagna. Vorogame: A hybrid p2p architecture for massively multiplayer games. In *Consumer Communications and Networking Conference (CCNC)*, pages 1–5, Jan 2009.

[53] Manuel Caroli, Pedro Machado Manhães de Castro, Sébastien Loriot, Olivier Rouiller, Monique Teillaud, and Camille Wormser. Robust and efficient delaunay triangulations of points on or close to a sphere. In *Experimental Algorithms, 9th International Symposium, SEA, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, pages 462–473, 2010.

[54] Manuel Caroli, Pedro Machado Manhães de Castro, Sébastien Loriot, Olivier Rouiller, Monique Teillaud, and Camille Wormser. Robust and efficient delaunay triangulations of points on or close to a sphere. In *Experimental Algorithms, 9th International Symposium, SEA*, pages 462–473, 2010.

[55] Jesús Carretero, Florin Isaila, Anne-Marie Kermarrec, François Taïani, and Juan M. Tirado. Geology: Modular georecommendation in gossip-based social networks. In *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012*, pages 637–646, 2012.

[56] Justin Clark Casey. Scaling opensimulator: An examination of possible architectures for an internet scale virtual environment network. *University of Oxford*, 2010.

[57] Alvin Chen and Richard R. Muntz. Peer clustering: a hybrid approach to distributed virtual environments. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

[58] Renjie Chen and Craig Gotsman. Localizing the delaunay triangulation and its parallel implementation. *Transactions on Computational Science*, 20:39–55, 2013.

[59] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB, 1997.

[60] Bruce Damer. Meeting in the ether: A brief history of virtual worlds as a medium for user-created events. *Journal For Virtual Worlds Research*, 1(1), 2008.

[61] Ville-Veikko Mattila David J. Murphy, Tuomas Vaittinen. Mirror worlds as large-scale outdoor mixed reality enablers. In *ISMAR*, 2011.

[62] Jean de Campredon. Un accès web au monde miroir hybridearth : Réalité hybride à l'échelle de la planète. *Rapport de stage, EURECOM*, 2013.

[63] Jauvane C. de Oliveira and Nicolas D. Georganas. Velvet: An adaptive hybrid architecture for very large virtual environments. *Presence*, 12(6):555–580, 2003.

[64] Thomas Debeauvais, Arthur Valadares, and Cristina Videira Lopes. RCAT: A restful client-scalable architecture. In *10th Annual Workshop on Network and Systems Support for Games, NetGames 2011, Ottawa, Ontario, Canada, October 6-7, 2011*, pages 1–2, 2011.

[65] Thomas Debeauvais, Arthur Valadares, and Cristina Videira Lopes. Evolution of scalability with synchronized state in virtual environments. In *Haptic Audio Visual Environments and Games (HAVE), 2012 IEEE International Workshop on*, 2012.

[66] Alexandre Denault, César Cañas, Jörg Kienzle, and Bettina Kemme. Triangle-based obstacle-aware load balancing for massively multiplayer games. In *10th Annual Workshop on Network and Systems Support for Games, NetGames 2011, Ottawa, Ontario, Canada, October 6-7, 2011*, pages 1–6, 2011.

[67] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a Triangulation. In *SoCG*, pages 106–114, New York, NY, USA, 2001. ACM.

[68] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. MOVIES: indexing moving objects by shooting index images. *GeoInformatica*, 15(4):727–767, 2011.

[69] C. du Mouza, W. Litwin, and P. Rigaux. SDR-tree: a Scalable Distributed Rtree. In *ICDE IEEE International Conference on Data Engineering*, 2007.

[70] Herman Arnold Engelbrecht and Gregor Schiele. Koekepan: Minecraft as a research platform. In *Annual Workshop on Network and Systems Support for Games, NetGames '13, Denver, CO, USA, December 9-10, 2013*, pages 1–3, 2013.

[71] Lu Fan, Philip W. Trinder, and Hamish Taylor. Design issues for peer-to-peer massively multiplayer online games. pages 108–125, 2010.

[72] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informtica*, 4:1–9, 1974.

[73] Luis Garcés-Erice, Ernst W. Biersack, Keith W. Ross, Pascal Felber, and Guillaume Urvoy-Keller. Hierarchical peer-to-peer systems. *Parallel Processing Letters*, 13(4):643–657, 2003.

[74] David Gelernter. Mirror worlds: or the day software puts the universe in a shoebox...how it will happen and what it will mean. pages I–XII, 1–237, 1992.

[75] Marta C. Gonzalez, Cesar A. Hidalgo, and Albert-Laszlo Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, June 2008.

[76] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI)*, pages 22–22, 2000.

[77] Nitin Gupta, Alan Demers, Johannes Gehrke, Philipp Unterbrunner, and Walker White. Scalability for virtual worlds. In *ICDE*, pages 1311–1314, 2009.

[78] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, 1984.

[79] Ellis Hamburger. Largest space battle in history claims 2,900 ships, untold virtual lives. http://www.theverge.com, July 2013.

[80] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. VON: A Scalable Peer-to-Peer Network for Virtual Environments. *IEEE Network Journal*, July 2006.

[81] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. In *Network and System Support for Games*, pages 129–133, 2004.

[82] H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *International Conference on Data Engineering (ICDE)*, page 34, 2006.

[83] Joaquín Keller and Gwendal Simon. Toward a Peer-to-Peer Shared Virtual Reality. In *ICDCS*, number July, pages 695–700, Washington, DC, USA, 2002. IEEE Comp. Soc.

[84] Joaquín Keller and Gwendal Simon. Solipsis: A massively multi-participant virtual world. In *PDPTA*, pages 262–268, 2003.

[85] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.

[86] Linden Lab. Oculus rift dk2 project viewer now available. http://community.secondlife.com/t5/Featured-News/Oculus-Rift-DK2-Project-Viewer-Now-Available/ba-p/2843450. Retrieved November 30, 2014.

[87] Dan Lake, Mic Bowman, and Huaiyu Liu. Distributed scene graph to enable thousands of interacting users in a virtual environment. In *NETGAMES*, pages 1–6, 2010.

[88] Sergey Legtchenko, Sébastien Monnet, and Gaël Thomas. Blue banana: resilience to avatar mobility in distributed mmogs. In *Dependable Systems & Networks (DSN)*, 2010.

[89] Huiguang Liang, Ransi Nilaksha De Silva, Wei Tsang Ooi, and Mehul Motani. Avatar mobility in user-created networked virtual worlds: measurements, analysis, and implications. pages 163–190, 2009.

[90] Huaiyu Liu, Mic Bowman, Robert Adams, John Hurliman, and Dan Lake. Scaling virtual worlds: Simulation requirements and challenges. In *Winter Simulation Conference*, pages 778–790, 2010.

[91] Cristina Videira Lopes. Hypergrid: Architecture and protocol for virtual world interoperability. *IEEE Internet Computing*, 15(5):22–29, 2011.

[92] Ingo Lütkebohle and Jacob Granberry. Westeroscraft: Home. http://westeroscraft.com. Retrieved September 2, 2014.

[93] Miguel Matos, Pascal Felber, Rui Oliveira, José Orlando Pereira, and Etienne Riviere. Scaling up publish/subscribe overlays using interest correlation for link sharing. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2462–2471, 2013.

[94] John L. Miller and Jon Crowcroft. Probabilistic event resolution with the pairwise random protocol. In *NOSSDAV 2009, Williamsburg, VA, USA. June 3-5, 2009, Proceedings*, pages 67–72, 2009.

[95] Katherine L. Morse, Lubomir Bic, and Michael B. Dillencourt. Interest management in large-scale virtual environments. pages 52–68, 2000.

[96] Mahdi Tayarani Najaran, Shun-Yun Hu, and Norman C. Hutchinson. SPEX: scalable spatial publish/subscribe for distributed virtual worlds without borders. In *Multimedia Systems Conference 2014, MMSys '14, Singapore, March 19-21, 2014*, pages 127–138, 2014.

[97] Simon Parkin. The Secret to a Video-Game Phenomenon. *MIT Technology Review*, 116(4), June 2013.

[98] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. Soccer video and player position dataset. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, pages 18–23, New York, NY, USA, 2014. ACM.

[99] Daniel Pittman and Chris GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NETGAMES*, pages 25–30, 2007.

[100] Daniel Pittman and Chris GauthierDickey. Characterizing virtual populations in massively multiplayer online role-playing games. In *MMM*, pages 87–97, 2010.

[101] Hernan Rozenfeld, Diego Rybski, Xavier Gabaix, and Hernan A. Makse. The area and population of cities: New insights from a different perspective on cities. 2011.

[102] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[103] Neal Stephenson. *Snow crash*. Bantam Books, 1992.

[104] Chris Stokel-Walker. Second life's strange second life. http://www.theverge.com, September 2013.

[105] Elodie Nilane Triponez. Hybrid earth: Mixed reality at planet scale. *EPFL Computer Science Master's project*, 2013.

[106] Mathieu Valero, Luciana Arantes, Maria Gradinariu, and Pierre Sens. Dynamically reconfigurable filtering architectures. In *SSS, New York, NY, USA, September 20-22, 2010. Proceedings*, pages 504–518, 2010.

[107] Matteo Varvello, C. Diot, and Ernst W. Biersack. P2P second life: Experimental validation using kad. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 1161–1169, 2009.

[108] Matteo Varvello, Christophe Diot, and Ernst Biersack. A walkable kademlia network for virtual worlds. In *Proceedings of the 8th International conference on Peer-to-peer systems, IPTPS, Boston, MA, USA, April 21*, page 2, 2009.

[109] Matteo Varvello, Stefano Ferrari, Ernst Biersack, and Christophe Diot. Exploring second life. *IEEE/ACM Transactions on Networking*, 19(1):80–91, 2011.

[110] Matteo Varvello, Stefano Ferrari, Ernst W. Biersack, and Christophe Diot. Distributed avatar management for second life. In *8th Annual Workshop on Network and Systems Support for Games, NetGames 2009, Paris, France, 23-24 November, 2009*, pages 1–6, 2009.

[111] Vernor Vinge. *Rainbows end*. Tor Science Fiction, 2007.

[112] Alexei Vázquez, João Gama Oliveira, Zoltán Dezsö, Kwang-Il Goh, Imre Kondor, and Albert-László Barabási. Modeling bursts and heavy tails in human dynamics. *Physical Review E*, 73(3):036127, March 2006.

[113] David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering*, ICDE, 1996.

[114] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9, 2013.

[115] Beverly Yang and Hector Garcia-Molina. Designing a Super-Peer Network. In *International Conference on Data Engineering (ICDE)*, pages 49–60, 2003.

[116] Anthony (Peiqun) Yu and Son T. Vuong. Mopar: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV*, pages 99–104, 2005.